Convergent Dual Bounds Using an Aggregation of Set-Covering Constraints for Capacitated Problems

Daniel Cosmin Porumbel, François Clautiaux

Convergent Dual Bounds Using an Aggregation of Set-Covering Constraints for Capacitated Problems

Daniel Cosmin Porumbel[1], François Clautiaux[2]

[1]Université Lille Nord de France, Université d'Artois, LGI2A, F-62400, Béthune, France

[2] Université Bordeaux 1, Institut de Mathématiques de Bordeaux, INRIA Bordeaux Sud-Ouest

Abstract

Extended formulations are now widely used to solve hard combinatorial optimization problems. Such formulations have prohibitively-many variables and are generally solved via Column Generation (CG). CG algorithms are known to have frequent convergence issues, and, up to a sometimes large number of iterations, classical Lagrangian dual bounds may be weak. This paper is devoted to set-covering problems in which all elements to cover require a given *resource consumption* and all feasible configurations have to verify a *resource constraint*.

We propose an iterative aggregation method for determining convergent dual bounds using the extended formulation of such problems. The set of dual variables is partitioned into k groups and all variables in each group are artificially linked using the following groupwise restriction: the dual values in a group have to follow a linear function of their corresponding resource consumptions. This leads to a restricted model of smaller dimension, with only $2k$ dual variables. The method starts with one group ($k = 1$) and iteratively splits the groups. Our algorithm has three advantages: (i) it produces good dual bounds even for low k values, (ii) it reduces the number of dual variables, and (iii) it may reduce the time needed to solve sub-problems, in particular when dynamic programming is used.

We experimentally tested our approach on two variants of the cutting-stock problem: in many cases, the method produces near optimal dual bounds after a small number of iterations. Moreover the average computational effort to reach the optimum is reduced compared to a classical column generation algorithm.

1 Introduction

Column Generation (CG) is a well known technique for solving Linear Programs (LPs) with prohibitively-many decision variables. At each iteration, a CG method considers a Reduced Master Problem (RMP) with a limited number of variables. This RMP is solved to optimality so as to generate: (i) a primal bound for the CG optimum, and (ii) dual values that are provided as an input of a pricing sub-problem that computes a new variable of attractive reduced cost. By iteratively solving the sub-problem, columns are added to the RMP and the primal bounds (RMP optima) converge toward the original CG optimum. A dual bound can be determined at each iteration by computing the Lagrangian bound associated to the current dual multipliers.

A frequent issue of column generation methods concern their convergence and (lack of) stabilization. In numerous cases, a large number of iterations might be needed to obtain a useful dual bound. The last decades have seen a surge of interest in stabilization methods that aim at decreasing the number of iterations in CG [20, 9, 17, 6, 22, 4, 8].

Aggregation and disaggregation techniques have a long history in optimization [25]. Generally speaking, the goal is to transform programs with high degree of detail into more coarser programs of smaller size. For instance, one can aggregate close time instants [19], nearby locations, related scheduling tasks, etc.; numerous examples and references are available in [16]. In the remaining, we focus however on the specific context of column generation. Constraint aggregation is used in

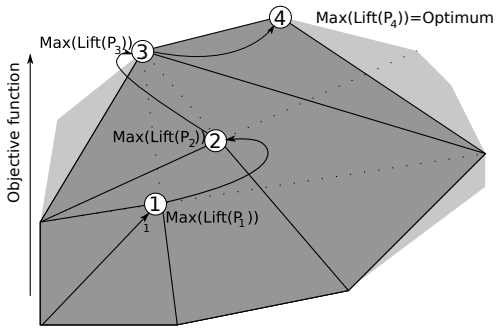

Figure 1: Our convergent method uses an "inner approximated dual polytope" that is iteratively refined ($\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, ...) so as to reach the optimum of an initial polytope $\mathcal{P}$ with prohibitively many constraints. The optimum of any intermediate $\mathcal{P}_k$ is a lower (dual) bound for the $\mathcal{P}$ optimum.

column generation for stabilization reasons, to reduce the number of dual variables, or to produce smaller RMPs with less degeneracy [10, 11, 3].

We propose a new aggregation technique sharing certain similarities with the Dynamic Constraint Aggregation (DCS) method [10, 11], and, implicitly, with the Stabilized DCS (SDCS) method [3] which combines DCS with a penalty-based stabilization technique [22]. To construct new columns, DCS first disaggregates the dual values to obtain a dual solution expressed in the original dual space (see also Section 2.2). Then, the initial subproblem is solved and the resulting column is either added to the current aggregated RMP, or put aside if it is not compatible with an associated partition. In our method, we decide a priori how the dual variables will be disaggregated by adding cuts that link the dual variables in each group. We add to the RMP all columns that are computed by the aggregated subproblem and some of the resulting columns are only feasible in the aggregated model. This may lead to non-valid dual cuts with respect to the original dual polytope, *i.e.*, excessively strong constraints. In other words, while DCS aims at reducing the number of iterations needed to converge while staying primal feasible, our method aims at producing useful iterative dual bounds by computing dual feasible solutions.

Such non-valid dual cuts generate, for each given number of groups k, an *inner* approximation $\mathcal{P}_k$ of the dual polytope $\mathcal{P}$. This approximated polytope $\mathcal{P}_k$ has only two variables per group, and leads to an easier optimization problem when k is small. Once an optimal solution for $\mathcal{P}_k$ is found, a valid dual bound is obtained for the original problem. The method converges toward the optimum by iteratively breaking groups until the optimum of the aggregated model corresponds to the optimum in the original dimension of $\mathcal{P}$.

Fig. 1 presents an intuitive illustration of this convergent method. It constructs fast effective dual bounds in the first steps (coarse aggregation for small k) and performs incremental iterative calculations to progressively increase k. Compared to other aggregation methods, our approach has the advantage of providing valid dual bounds before fully converging, *i.e.*, each $\mathcal{P}_k$ optimum represents a valid dual bound for the original problem.

In this paper, we focus on set-covering master problems and pricing sub-problems modeled as Integer Linear Programs (ILPs) with a *resource constraint*. In these models, each element to cover requires a resource consumption. Our new dual *groupwise-linearity restrictions* work as follows: in each of the k groups, the dual values have to follow a linear function of corresponding resource consumptions. This allows us to reformulate the simplified problem with a new model whose variables are the slope and y-intercept of the linear function over each group. The motivation for using such aggregation comes from the fact that dual values are often correlated with the resource consumption at optimality.

The remainder is organized as follows. Section 2 presents the classical column generation method and a review of the literature. Section 3 is devoted to the convergent method: the first

part (§3.1-§3.3) describes the aggregation modelling for a fixed k and the second part presents the *incremental* calculations of the $\mathcal{P}_1, \mathcal{P}_2, \ldots$ sequence (§3.4-§3.6). In Section 4, we discuss the theoretical benefits of using groupwise-linearity constraints instead of simpler equality constraints. Section 5 specifies the details of an application of the method to the cutting-stock problem. Section 6 presents our computational experiments, followed by conclusions in the last section.

2 Context, Motivation and Literature Review

2.1 Column Generation for Set-Covering Formulations with Resource Constraints in the Subproblem

We first introduce the set-covering model that is considered in this paper. Such models are often used to solve, among many others, cutting and packing [13, 14], vehicle-routing [21], and employee scheduling [11, 10] problems.

The primal ILP is defined on a ground set of elements $I = \{1, \ldots, n\}$. A column, also called configuration hereafter, of the master problem is an integer column vector $\mathbf{a} = [a_1 \ldots a_n]^\top \in \mathbb{Z}_+^n$. The set $\mathcal{A}$ of all configurations is defined by all the solutions of a given sub-problem. The set-covering problem requires finding the minimum number of configurations that need to be selected so as to cover each element $i \in [1..n]$ at least b_i times ($b_i \in \mathbb{Z}_+$). To each configuration $\mathbf{a} \in \mathcal{A}$, we associate a variable λ_a that indicates the number of times configuration $\mathbf{a}$ is selected. This leads to the following ILP:

$$\begin{array}{ll} \min \sum_{\mathbf{a}\in\mathcal{A}} \lambda_a & \\ \sum_{\mathbf{a}\in A} a_i \lambda_a \geq b_i, & \forall i \in 1, \ldots, n \\ \lambda_a \in \mathbb{N}, & \forall \mathbf{a} \in \mathcal{A} \end{array} \tag{2.1}$$

We are interested in the linear relaxation of this model, where the last constraint is replaced by $\lambda_a \geq 0, \forall a \in \mathcal{A}$. The dual of this linear relaxation can be written using a vector $\mathbf{y} = [y_1\ y_2\ \ldots\ y_n]^\top \in \mathbb{R}_+^n$ of dual variables and a possibly exponential number of dual constraints.

$$\left.\begin{array}{ll} \max \mathbf{b}^\top \mathbf{y} & \\ \mathbf{a}^\top \mathbf{y} \leq 1, & \forall \mathbf{a} \in \mathcal{A} \\ y_i \geq 0, & i \in 1, \ldots, n \end{array}\right\} \mathcal{P} \tag{2.2}$$

In the remainder, we will refer to this problem as the *Dual Set-Covering Problem* (DSCP) over polytope $\mathcal{P}$: $\text{DSCP}(\mathcal{P}) = \max\{\mathbf{b}^\top \mathbf{y} : \mathbf{y} \in \mathcal{P}\}$. The optimum of this program, hereafter noted $\text{OPT}(\text{DSCP}(\mathcal{P}))$, represents the main focus of all methods from this paper. It is also termed the CG optimum OPT_{CG}, or the optimum over $\mathcal{P}$ on objective function $\mathbf{b}$ (we sometimes omit mentioning $\mathbf{b}$, as we never use other functions).

Given the prohibitive large size of $\mathcal{A}$, this constraint set is generated iteratively. From a dual point of view, the classical column generation can be seen as a cutting-plane algorithm: (i) reach the best dual solution in the current dual polytope defined by the limited set of constraints that have been generated so far, (ii) find a valid dual constraint that is violated by the current solution (report optimum if there is no such violated constraint), and (iii) refine the current formulation by adding a violated dual constraint and repeat from (i). Therefore, the column generation method constructs at each step an *outer* approximation of the dual polytope that is optimized.

The pricing sub-problem in CG requires finding a configuration of most negative reduced cost:

$$\min_{\mathbf{a}\in\mathcal{A}} \left(1 - \mathbf{y}^\top \mathbf{a}\right) = 1 - \max_{\mathbf{a}\in\mathcal{A}} \mathbf{y}^\top \mathbf{a} \tag{2.3}$$

In the following, we will forget the constant and seek a configuration $\mathbf{a}^*$ that maximizes $\mathbf{y}^\top \mathbf{a}^*$. Note that in this sub-problem, $\mathbf{y}$ represents input data; at each iteration, $\mathbf{y}$ is the dual optimum solution of the RMP.

A key point in column generation is the speed of the sub-problem solution method. Dynamic Programming (DP) algorithms provide an efficient tool for generating columns for many different families of problems (see among many others [21, 13]). We now formalize a general DP framework for problems with a single resource constraint. We consider a resource vector $\mathbf{w} = [w_1 \ w_2 \dots w_n]^\top \in \mathbb{R}^n_+$ and a lower and upper limit of total resource consumption $(C^-, C^+ \in \mathbb{N})$. For a given dual solution $\mathbf{y} \in \mathbb{R}^n$, the sub-problem requires selecting a_i times each element $i \in [1..n]$ (*i.e.*, finding a configuration) so as to maximize the total profit $\mathbf{y}^\top \mathbf{a}$ under the resource constraint: $C^- \leq \mathbf{w}^\top \mathbf{a} \leq C^+$.

The dynamic programming recursion computes a profit function $P_{\max}$ that maps any state $(c, i) \in [0, C^+] \times [1..n]$ to the maximum profit $P_{\max}(c, i)$ that can be obtained with a resource amount of c, only selecting elements of $[1..i]$. Generally speaking, this function can be determined using a recursion of the form below.

$$P_{\max}(c, i) = \max \Big\{ P_{\max}(c - r \cdot w_i, i-1) + r \cdot y_i : \ r \in [0..b_i], \ r \cdot w_i \leq c \Big\} \tag{2.4}$$

The initial state is $P_{\max}(0, 0) = 0$ and the final minimum reduced cost is attained in a state (c^*, n) such that $P_{\max}(c^*, n) = \max_{c \in [C^-, C+]} P_{\max}(c, n)$.

If the given capacitated problem imposes additional conditions on feasible patterns, this formula is generalized by imposing additional constraints on the feasible subsets of elements. For example, in *Bin Packing with conflicts* [12], some pairs of elements cannot be selected together.

The above recursion can be computed in $O(n_b C^+)$ time, where $n_b = \sum_{i=1}^n b_i$ is the number of *individualized elements*. We use n_b instead of n: the elements with demand multiplicities $b_i > 1$ are not considered only once in the DP scheme, but b_i times—observe $r \in [0..b_i]$ in (2.4). For the *Bin-Packing problem*, $n_b = n$ and the complexity becomes $O(nC^+)$. In other resource-constrained problems, such as the vector-packing problem [5], additional constraints can restrict the set of feasible patterns and/or increase the dimension of the state space.

Although we describe here the case with one resource, it can also be applied when several resources are considered. One only needs multiple dimensions to index the states.

2.2 Aggregation in Column Generation and Extended Formulations

A way to cope with prohibitively large Mixed-Integer Programs (MIPs) arising in extended formulations is to approximate them. This can be done by restricting the MIP to a subset of variable and/or constraints. This leads to a primal inner approximation that can be refined iteratively (typically by column-and-row generation, see [23] for a generic view of those methods).

Another way of obtaining a tractable model is to apply aggregation to the constraints or to the variables. This is done statically by [28], who define a smaller model whose size depends on a given parameter. An interesting conclusion of [28] is that small values of this parameter are often sufficient to obtain excellent bounds. However this method is static and does not converge toward the optimum of the initial model.

In the context of column generation methods, similar aggregations can be obtained by adding equality constraints on dual variables of a same cluster. This has the advantage to reduce the size of the pricing sub-problem (since all dual variables of the cluster can be exchanged), and also to stabilize the column generation process. This strategy is useful if many dual variables are "equivalent" and thus many of them can have similar values at optimality. Dual optimal and deep dual optimal inequalities [1] are examples of dual constraints that do not separate all optimal dual solutions.

Dynamic constraint aggregation has been introduced in the context of column generation in [11, 10, 1]. In the aggregated model, each constraint represents a cluster of original partitionning constraints. From a dual space perspective, a dual variable of the aggregated model represents a cluster of original dual variables, and therefore its value is equal to the sum of the actual dual values in the cluster. When the pricing subproblem is called, the dual variables are disaggregated and the initial subproblem is solved. The column produced by the subproblem is added to the RMP if it is compatible with the current aggregation, or put aside otherwise. These methods ensure the

exactness of the solution by iteratively breaking up the clusters until an optimal feasible solution is obtained.

In the above approach, a compatible column corresponds to a configuration that either covers all elements of a given group or none of them. The elements of a given group are *not* distinguished in the aggregated model, *i.e.*, they are in the same equivalence class and they are all associated to a unique representative partitioning constraint. For some specific problems, such as the *cutting-stock problem*, we are given items with different weights (resource consumptions); if the weights of two items are different, they might have significantly different dual values at optimality, and so, they can hardly be considered equivalent. In such situations, equivalence-based approaches might require too many (small) clusters to obtain good approximations. An extreme case is obtained when each cluster must contain exactly one element, even if the structure of the solution is simple (for example the dual values can be obtained by applying a linear function to the item sizes).

In certain cases, the dual values may often follow a piecewise-linear function of the associated resource consumptions. This type of solution can be obtained by a new type of aggregation, where dual variables in the same cluster have values that follow a linear function of this resource. This is what we intend to do in the remainder of this paper. The basic underlying idea is that elements requiring higher resource consumptions arise in less valid configurations, and so, they can be associated to larger dual values; for cutting-stock, the dual solution vector is always non-decreasing (see for example [6]).

3 Iterative Inner Dual Approximation (2IDA)

This section describes the iterative aggregation method, starting from a general overview and gradually addressing all technical details. From now on, we will now focus on the the dual polytope $\mathcal{P}$, *i.e* we consider CG as a cutting-plane algorithm for optimizing (2.2).

3.1 Full Description of the Aggregated Dual Polytope

Let $G_k = \{I^1, I^2, \dots I^k\}$ be a partition of $I = \{1, 2 \dots n\}$ into k groups, each group having $n_j = |I^j|$ elements. Without loss of generality, we consider that the elements in each group are sorted (indexed) in increasing resource consumption. Given elements I^j of a group $j \in [1..k]$, let $\mathbf{y}^j$, $\mathbf{w}^j$, $\mathbf{b}^j$ and $\mathbf{a}^j$ denote the n_j-dimensional column vectors related to dual variables, resource consumptions, demands and, respectively, coefficients of some configuration $\mathbf{a} \in \mathcal{A}$. Variables in the dual vector are ordered in such a way that $\mathbf{y}$ is the concatenation of k group components: $\mathbf{y}^\top = [y_1\ y_2 \dots y_n] = [(\mathbf{y}^1)^\top \dots (\mathbf{y}^k)^\top]$.

We artificially impose the following groupwise-linearity restrictions: given any group $j \in [1..k]$, the values of the dual variables $y^j_1, y^j_2, \dots y^j_{n_j}$ have to follow a linear function of the resource consumption of the corresponding element. This function depends on a slope α^j and a y-intercept β^j. More precisely, the i^{th} component of $\mathbf{y}^j$ can be written as $y^j_i = w^j_i \alpha^j + \beta^j$. By simply adding these constraints to (2.2), we obtain the *inner dual polytope*:

$$
\begin{array}{lll}
 & \max \mathbf{b}^\top \mathbf{y} & \\
\mathbf{a}^\top \mathbf{y} \leq 1, & \forall \mathbf{a} \in \mathcal{A} & \\
y^j_i = w_i \alpha^j + \beta^j, & \forall j \in [1..k], i \in I^j & (3.1) \\
y^j_i \geq 0, & \forall j \in [1..k], i \in I^j & \\
\alpha^j, \beta^j \in \mathbb{R}, & \forall j \in [1..k] &
\end{array}
$$

Remark 1. *By extracting the vector* $\mathbf{y}$ *from any valid solution of above* (3.1)*, we obtain a valid solution of* $\mathcal{P}$ *in* (2.2)*. The opposite is not true in general: only vectors* $\mathbf{y}$ *of* $\mathcal{P}$ *with the suitable groupwise linear structure can be lifted to valid solutions of* (3.1)*.*

Let us re-write (3.1) using only the new decision variables α^j and β^j. We first re-write the objective function. For each group j, $\mathbf{y}^j$ can be written as a linear combination of $\mathbf{w}^j$ and $\mathbf{1}_j$

(vector $[1\ 1 \dots\ 1]^\top$ with n_j elements): $\mathbf{y}^j = \mathbf{w}^j\alpha^j + \mathbf{1}_j\beta^j$. Observe that

$$\mathbf{b}^\top\mathbf{y} = \sum_{j=1}^{k} \left(\mathbf{b}^j\right)^\top \mathbf{y}^j = \sum_{j=1}^{k} \left(\mathbf{b}^j\right)^\top \left(\mathbf{w}^j\alpha^j + \mathbf{1}_j\beta^j\right) = \sum_{j=1}^{k} \left(\left(\mathbf{b}^j\right)^\top \mathbf{w}^j\right)\alpha^j + \left(\left(\mathbf{b}^j\right)^\top \mathbf{1}_j\right)\beta^j \quad (3.2)$$

The first set of constraints from (3.1) can be written $\sum_{j=1}^{k} \left(\mathbf{a}^j\right)^\top \mathbf{y}^j \leq 1, \mathbf{a} \in \mathcal{A}$. For each j, we have:

$$\left(\mathbf{a}^j\right)^\top \mathbf{y}^j = \left(\mathbf{a}^j\right)^\top \left(\mathbf{w}^j\alpha^j + \mathbf{1}_j\beta^j\right) = \left(\left(\mathbf{a}^j\right)^\top \mathbf{w}^j\right)\alpha^j + \left(\left(\mathbf{a}^j\right)^\top \mathbf{1}_j\right)\beta^j \quad (3.3)$$

We are now ready to express model (3.1) with variables α^j and β^j only. To simplify the notation, we introduce the following definitions.

Definition 1. *Given a configuration $\mathbf{a} \in \mathcal{A}$ and a group j, we define:*

- $c_a^j = \left(\mathbf{a}^j\right)^\top \mathbf{w}^j$*: total resource consumption of the elements of I^j selected in $\mathbf{a}$. Remark this is the coefficient of variable α^j in* (3.3)*;*
- $N_a^j = \left(\mathbf{a}^j\right)^\top \mathbf{1}_j$*: total number of elements of I^j selected in $\mathbf{a}$. This is the coefficient of variable β^j in* (3.3)*;*
- $w^j_{\min}$ *and* $w^j_{\max}$*: the lowest and respectively highest resource consumption of an element of group j.*

By simply substituting (3.2)-(3.3) in model (3.1), after handling the non-negativity constraints, we obtain an equivalent model in the space $\mathbb{R}^{2k}$:

$$\left.\begin{array}{ll} \max \sum_{j=1}^{k} \left(\left(\mathbf{b}^j\right)^\top \mathbf{w}^j\right)\alpha^j + \left(\left(\mathbf{b}^j\right)^\top \mathbf{1}_j\right)\beta^j & \\ \sum_{j=1}^{k} c_a^j\alpha^j + N_a^j\beta^j \leq 1, & \forall \mathbf{a} \in \mathcal{A} \\ w^j_{\min}\alpha^j + \beta^j \geq 0 & \forall j \in [1..k] \\ w^j_{\max}\alpha^j + \beta^j \geq 0 & \forall j \in [1..k] \\ \alpha^j, \beta^j \in \mathbb{R}, & \forall j \in [1..k] \end{array}\right\} \mathcal{P}_k \quad (3.4)$$

Proposition 1. *There is a bijection between the set of feasible values of $[y_1\ y_2\ \dots y_n]$ in the model (3.1) in $\mathbb{R}^n_+ \times \mathbb{R}^{2k}$ and the feasible solutions $[\alpha^1 \dots \alpha^k\ \beta^1 \dots \beta^k]^\top$ of model (3.4) in $\mathbb{R}^{2k}$.*

Proof. Recall that (3.1) has by construction the constraint $y_i^j = w_i\alpha^j + \beta^j$. The bijection simply maps any solution $[\mathbf{y}^1 \dots \mathbf{y}^n, \alpha^1 \dots \alpha^k, \beta^1 \dots \beta^k]^\top \in \mathbb{R}^n_+ \times \mathbb{R}^{2k}$ that is valid in (3.1) to a solution $[\alpha^1 \dots \alpha^k, \beta^1 \dots \beta^k]^\top \in \mathbb{R}^{2k}$ that is valid in (3.4). Recall (3.4) is simply obtained by replacing all y_i^j terms of (3.1) with $y_i^j = w_i\alpha^j + \beta^j$, using (3.3) and (3.2). The non-negativity constraints of (3.1) are simplified in (3.4), *i.e.*, $y_i^j = w_i^j\alpha^j + \beta^j \geq 0, \forall j \in [1..k], i \in I^j$ is satisfied if $y^j_{\min} = w^j_{\min}\alpha^j + \beta^j \geq 0$ and $y^j_{\max} = w^j_{\max}\alpha^j + \beta^j \geq 0$: the value $w_i^j\alpha^j$ is always between $w^j_{\min}\alpha^j$ and $w^j_{\max}\alpha^j$. □

The new formulation (3.4) has $2k$ real variables that can be either *positive* or *negative*. In this basic version, only the number of variables is decreased. This reduction can be useful for stabilization reasons in CG: using less dual variables, one reduces the possibility of oscillation. Also, less dual variables corresponds to less constraints in the corresponding Reduced Master Problems (RMPs), which can accelerate the LP solver for RMPs.

However, a drawback of (3.4) consists of the fact that it has the same (prohibitively large) number of constraints as (2.2). We address this point in the next section, introducing a new model (3.5) that removes many constraints that become redundant in (3.4).

3.2 Column Generation for the Agregated Set-Covering Problem

When one imposes linearity restrictions on the dual variables, many configurations of $\mathcal{A}$ are not useful in (3.4) anymore. This will be shown in Section 3.2.1 below, leading to a new model (3.5) that reduces the size of the initial set-covering LP in terms of both columns and rows. This later model is the one which is actually optimized by our method; more computational and optimization discussion follow in Section 3.2.2.

3.2.1 Removing Redundant Constraints in Model (3.4)

Model (3.4) does not explicitly indicate the non-zero elements in a configuration $\mathbf{a} \in \mathcal{A}$. The main constraints of (3.4) are built only by considering the number of elements of each group N_a^j and the total resource consumption c_a^j. Therefore, to avoid redundancies, we no longer express configurations as vectors in $\mathbb{R}_+^n$, but as aggregated $\mathbb{R}^{2k}$ vectors of the form $\overline{\mathbf{a}} = [c_a^1\ N_a^1, c_a^2\ N_a^2, \ldots, c_a^k\ N_a^k]$. We say that the aggregated configuration $\overline{\mathbf{a}}$ can be lifted to a configuration $\mathbf{a} \in \mathcal{A}$ if:

1. $C^- \leq \sum_{j=1}^k c_a^j, \leq C^+$, *i.e.*, the total consumption is feasible;
2. there exist $\mathbf{a} \in \mathcal{A}$ such that $c_a^j = \left(\mathbf{a}^j\right)^\top \mathbf{w}^j$ and $N_a^j = \left(\mathbf{a}^j\right)^\top \mathbf{1}_j$, for all $j \in [1..k]$.

There might exist more than one disaggregated configuration $\mathbf{a} \in \mathcal{A}$ that corresponds to an aggregated configuration $\overline{\mathbf{a}}$, *i.e.*, more configurations $\mathbf{a}$ might have the same values $c_a^j,\ N_a^j,\ \forall j \in [1..k]$. Therefore, many constraints of $\mathcal{P}_k$ in (3.4) are redundant.

Definition 2. *Given group $j \in [1..j]$, the set R^j of feasible resource consumptions is:*

$$R^j = \left\{c^j \in [0..C^+] :\ \exists \mathbf{a} \in \mathcal{A} \text{ such that } c^j = c_a^j\right\}$$

Definition 3. *Given a feasible consumption $c^j \in R^j$, we define the minimum and maximum cardinality coefficients of c^j for group j:*

- *$N^+(j, c^j)$: the maximum number of I^j elements in a valid configuration $\mathbf{a} \in \mathcal{A}$ in which the total resource consumption of the I^j elements is c^j;*
- *$N^-(j, c^j)$: the minimum number of I^j elements in a valid configuration in which the total resource consumption of the I^j elements is c^j.*

Definition 4. *Let $\mathcal{A}_k$ be the set of dominant (non-redundant) configurations, defined by:*

$$\mathcal{A}_k = \left\{\mathbf{a} \in \mathcal{A},\ \text{such that } \forall j \in [1..j], N_a^j \in \left\{N^+(j, c_a^j),\ N^-(j, c_a^j)\right\}\right\}$$

By replacing $\mathcal{A}$ with $\mathcal{A}_k$ in model (3.4), we obtain a new model:

$$\left.\begin{array}{ll} \max \sum_{j=1}^k \left(\left(\mathbf{b}^j\right)^\top \mathbf{w}^j\right) \alpha^j + \left(\left(\mathbf{b}^j\right)^\top \mathbf{1}_j\right) \beta^j & \\ \sum_{j=1}^k c_a^j \alpha^j + N_a^j \beta^j \leq 1, & \forall \mathbf{a} \in \mathcal{A}_k \\ w_{\min}^j \alpha^j + \beta^j \geq 0 & \forall j \in [1..k] \\ w_{\max}^j \alpha^j + \beta^j \geq 0 & \forall j \in [1..k] \\ \alpha^j, \beta^j \in \mathbb{R} & \forall j \in [1..k] \end{array}\right\} \mathcal{P}_k \tag{3.5}$$

Obviously, if two configurations $\mathbf{a}, \mathbf{a}' \in \mathcal{A}$ lead to the same values of N_a^j and c_a^j, only one constraint has to be explicitly considered. The new optimization problem, will be hereafter referred to as the *Dual Set-Covering Problem* (DSCP) over polytope $\mathcal{P}_k$: $\text{DSCP}(\mathcal{P}_k) = \max\{\sum_{j=1}^k ((\mathbf{b}^j)^\top \mathbf{w}^j)\alpha^j + ((\mathbf{b}^j)^\top \mathbf{1}_j)\beta^j : [\alpha^1 \ldots \alpha^k, \beta^1 \ldots \beta^k]^\top \in \mathcal{P}_k\}$. We will show below that (3.5) determines the same polytope $\mathcal{P}_k$ as (3.4), and so, any solution of (3.5) can be lifted to a solution of the initial LP (2.2).

Proposition 2. *Any solution of* (3.5) *can be lifted to a solution that is feasible for the initial set-covering model* (2.2).

Proof. Remark 1 (p. 5) states that any solution of $\mathcal{P}_k$ in (3.1) can be projected into a feasible solution of the initial dual polytope $\mathcal{P}$ in (2.2). Proposition 1 shows that $\mathcal{P}_k$ is modelled equivalently by (3.1) and (3.4). It is enough to show that (3.4) is equivalent to (3.5).

Using the notation from Definitions 1 and 3, we observe that, given any valid $\mathbf{a} \in \mathcal{A}$, for each $j \in [1..k]$, the values of variables α^j and β^j have to respect one of the two inequalities below:

1. $c_a^j \alpha^j + N_a^j \beta^j \leq c_a^j \alpha^j + N^+(j, c_a^j)\beta^j$, if $\beta^j \geq 0$;
2. $c_a^j \alpha^j + N_a^j \beta^j \leq c_a^j \alpha^j + N^-(j, c_a^j)\beta^j$, if $\beta^j < 0$.

Therefore, the constraints obtained from configurations outside $\mathcal{A}_k$ are dominated by constraints associated to configurations $\mathbf{a} \in \mathcal{A}_k$, *i.e.*, with $N_a^j = N^+(j, c_a^j)$ or $N_a^j = N^-(j, c_a^j)$. All configurations outside $\mathcal{A}_k$ are dominated, and so, (3.4) is equivalent to (3.5). □

To solve DSCP($\mathcal{P}_k$), we need to determine all coefficients $N^+(j, c^j)$ or $N^-(j, c^j)$, $\forall c^j \in R^j, j \in [1..k]$. This is performed in a *coefficient calculation preprocessing stage* which is executed only once before launching column generation. The goal is to find the maximum and minimum number of elements that can be selected from I^j so as to consume a total resource amount of c^j, $\forall c^j \in \mathbb{R}^j, j \in [1..k]$. This task is similar to the pricing sub-problem (2.3) for the non-aggregated problem (Section 2.1). If the dynamic programming scheme for (2.4) is fast, so is the dynamic programming for this coefficient calculation. Indeed, it is enough to replace y_i with 1 in (2.4) and to apply an analogous dynamic programming twice, *i.e.*, once with a minimization and once with a maximization objective. Such a dynamic programming scheme generates by default a state for each feasible c^j value, *i.e.*, the set R^j. Considering all groups $j \in [1..k]$ together, this coefficient calculation stage requires the same complexity as *one pricing sub-problem* for the initial model (2.2).

Remark 2. *The number of constraints in* (3.5) *is exponentially large in* k*, but not in* n*. Computing all constraints of* $\mathcal{P}_k$ *would thus require an asymptotic running time that is exponentially large in* k *and polynomial in* C^+ *(to determine all* $N^+(j, c^j)$ *and* $N^-(j, c^j)$*). If we consider that* k *and* C *are bounded constants, it is possible to achieve the full construction of* $\mathcal{P}_k$ *in polynomial time. This would lead to a pseudo-polynomial dual bound for the initial combinatorial optimization problem. Although such direct* $\mathcal{P}_k$ *constructions are used in this paper only for* $k = 1$*, the main idea of a direct construction represents a promising subject for further work.*

3.2.2 An aggregated Pricing Subproblem

The model (3.5) is optimized by column generation for all values of $k > 1$. A computational difficulty is related to the pricing algorithm. In a straightforward approach, one could disaggregate the variables to obtain the dual solution $\mathbf{y}$ in the original space in $\mathbb{R}_+^n$; then, the pricing algorithm for the non-aggregated model (Section 2.1) can be used. This has the disadvantage of considering a solution space with many symmetries and redundancies, since all y_i^j with $i \in [1..n_j]$ in a group $j \in [1..k]$ correspond to a single pair of dual variables α^j and β^j.

We now show how the pricing can be solved without disaggregation. Iteratively, for each current solution $[\alpha^1 \dots \alpha^k, \beta^1 \dots \beta^k]^\top \in \mathcal{P}_k$, the column generation algorithm needs to solve an aggregated version of the sub-problem (2.3): find the *aggregated configuration* $\overline{\mathbf{a}} = [c_a^1\ N_a^1, c_a^2\ N_a^2, \dots, c_a^k\ N_a^k]$ that maximizes the profit value $\sum_{j=1}^k \alpha^j c_a^j + \beta^j N(j, c_a^j)$. As for the non-aggregated pricing (2.3), a column of negative reduced cost is identified whenever this profit is larger than 1. Formally, the pricing problem is the following:

$$\begin{array}{ll} \max & \sum_{j=1}^k \alpha^j c^j + \beta^j N(j, c^j) \\ \text{s.t.} & C^- \leq \sum_{j=1}^j c^j \leq C^+ \\ & N(j, c^j) = \left\{ \begin{array}{ll} N^+(j, c^j) & \text{if } \beta^j \geq 0 \\ N^-(j, c^j) & \text{if } \beta^j < 0 \end{array} \right. \forall j \in [1..k] \\ & c^j \in R^j, \forall j \in [1..k] \end{array} , \tag{3.6}$$

where R^j is the set of feasible resource consumptions (Definition 2). This set is determined during the preprocessing stage for coefficient calculation from Section 3.2.1, *i.e.*, R^j is implicitly determined at the same time with the values $N^+(j, c^j)$ and $N^-(j, c^j)$. Recall that the dynamic programming approach proposed for this coefficient calculation requires the same asymptotic running time as one pricing algorithm for the original problem (2.2), for all $j \in [1..k]$.

The variables $c^j \in R^j,\ \forall j \in [1..j]$ are actually the only decision variables in (3.6). They are sufficient to represent a solution for this pricing sub-problem, since any $N(j, c^j)$ term can be deduced from c^j and β^j. All β^j represent input data for the sub-problem, and so, we choose from the beginning to use either $N(j, c^j) = N^+(j, c^j)$ or $N(j, c^j) = N^-(j, c^j)$, depending on the sign of β^j. Let us denote $p(j, c) = \alpha^j c + \beta^j N(j, c)$ the potential profit that can be obtained with a resource amount of c for group j. The pricing problem (3.6) is reformulated:

$$\begin{array}{ll} \max & \sum_{j=1}^{k} \sum_{c \in R^j} p(j,c) x_{jc} \\ \text{s.t.} & C^- \leq \sum_{j=1}^{j} \sum_{c \in R^j} c^j x_{jc} \leq C^+, \\ & \sum_{c \in R^j} x_{jc} = 1, \forall j \in [1..k] \\ & x_{jc} \in \{0,1\}, \forall j \in [1..k], c \in R^j \end{array} \tag{3.7}$$

where x_{jc} is a new decision variable for group j, *i.e.*, it is equal to 1 if group j has a total resource consumption of c, and 0 otherwise.

The resulting aggregated pricing is a *multiple-choice* variant of the non-aggregated pricing from Section 2.1. For instance, if the non-aggregated problem is the knapsack problem, the aggregated one is the multiple-choice knapsack problem. Generally speaking, the non-aggregated DP approach from Section 2.1 can be extended to an aggregated Dynamic Program (DP) as follows. We associated a DP state to each total consumption value in $[1..C^+]$, and, for each level $j \in [1..k]$, one can scan all feasible choices (total consumptions values c^j) to generate (or update) other states. Furthermore, in certain cases, the aggregated DP only needs to perform these computations on two levels associated to the new sub-groups $j_1, j_2 \in [1..k]$ (see Section 3.6), which is fast in practice. However, in this aggregated pricing, the number $n_b = \sum_{i=1}^{n} b_i$ of *individualized elements* is no longer a factor in the asymptotic or practical running time, once the initial preprocessing (Section 3.2.1) is performed.

3.3 The Iterative Algorithm and the Incremental $\mathcal{P}_k$ Optimization

Section 3.2 described an aggregation-based column generation method that optimizes (3.5) for a given *fixed* partition $G_k = \{I^1, I^2, \ldots I^k\}$. The resulting value, hereafter noted lb_{G_k} (or simply lb_k when the exact structure of G_k is not essential) represents a lower bound for the sought OPT_{CG}. Algorithm 1 below presents the general steps of our Iterative Inner Dual Approximation (2IDA) method. The main idea is to iteratively break the groups into smaller subgroups and incrementally refine the description of $\mathcal{P}_k$ until a stopping condition is met.

Algorithm 1: Iterative Inner Dual Approximation

```
1  k ← 1, G_1 ← {I}
2  P_k ← detP_kCoefs(G_k)                    // preprocessing coefficient calculation (Section 3.2.1)
3  repeat
4  |  lb_{G_k} ← aggregCGoptim(P_k)          // Section 3.2 (or full P_k construction for k = 1)
5  |  ub_{G_k} ← upBound(P_k, lb_{G_k})      // optional upper bound (Section 3.5) of OPT_CG
6  |  if ⌈lb_{G_k}⌉ < ⌈ub_{G_k}⌉ then
7  |  |  G_{k+1} ← grpSplit(G_k)            // split a group and refine partition (Section 3.4)
8  |  |  P_{k+1} ← detP_kCoefs(G_{k+1} − G_k)  //determine coefficients for the new sub-groups
9  |  |  P_{k+1} ← inheritCnstr(P_k,G_{k+1})  // inherit P_k constraints into P_{k+1} (Section 3.6)
10 |  |  k ← k + 1
11 until (⌈lb_{G_k}⌉ = ⌈ub_{G_k}⌉) or a stopping condition is reached ;
```

The partition G_k of the dual variables is initialized with one group equal to I. Right after that (in Line 2), 2IDA applies the preprocessing stage from Section 3.2.1 to determine the coefficients (*e.g.*, $N^-(j_1, c^{j_1})$, $N^+(j_1, c^{j_1})$, etc.) required to describe all aggregated polytopes. Routine `aggregCGoptim`($\mathcal{P}_k$) uses partition G_k to restrict and project the dual polytope $\mathcal{P}$ to dimension $\mathbb{R}^{2k}$ and describe polytope $\mathcal{P}_k$ (3.5). This polytope has fewer constraints and fewer variables than $\mathcal{P}$. For $k > 1$, `aggregCGoptim`($\mathcal{P}_k$) relies on a column generation scheme that iteratively solves the aggregated pricing sub-problem from Section 3.2.2.

After determining lb_{G_k}, Algorithm 1 can continue by calculating an optional upper bound ub_{G_k} (routine `upBound`($\mathcal{P}_k, \text{lb}_{G_k}$), Line 5). For this, we propose (see Section 3.5) a method that performs disaggregated CG steps, using a polytope $\mathcal{P}^u$ that contains much fewer constraints than $\mathcal{P}$. While upper bounds are not mandatory in 2IDA, they can be useful for "tail cutting" (*i.e.*, to stop 2IDA if $\lceil \text{lb}_{G_k} \rceil = \lceil \text{ub}_{G_k} \rceil$). Furthermore, the specific upper bound described in Section 3.5 can also be more effective in guiding group split decisions (see below).

Section 3.4 presents several methods for taking group split decisions (call `grpSplit`(G_k) in Line 7). The goal is to determine a group $j^* \in [1..k]$ to be split into sub-groups j_1 and j_2, so as to progress from k to $k+1$. This decision can be taken using information related to the upper bound (if available), problem-specific data, or general ad-hoc indicators (group sizes, resource consumption spreads, etc.). After calling `grpSplit`(G_k), 2IDA applies (in Line 8) the coefficient calculation preprocessing stage (Section 3.2.1), but only restricted to the new sub-groups j_1 and j_2.

The configurations (constraints) generated for a given k are not completely removed when 2IDA passes to $k+1$. The general goal of 2IDA is to perform an incremental construction of $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3 \ldots$. Instead of constructing each $\mathcal{P}_{k+1}$ from scratch, 2IDA picks up constraints generated when optimizing over $\mathcal{P}_k$ and lifts them in the new dimension of $\mathcal{P}_{k+1}$. In fact, routine `inheritCnstr`(P_k,G_{k+1}) make use of $\mathcal{P}_k$ constraints to warm-start the optimization over DSCP($\mathcal{P}_{k+1}$)—see Section 3.6 for a description in greater detail of this "inheritance" process.

3.4 Split Decisions

2IDA converges towards the optimum of the initial problem for any group split decisions. However, from a practical perspective, the split decision is a crucial part for the efficiency of the method. In many optimization settings, a fast convergence is obtained by making at each step a decision that leads to the best local improvement. However, the split decision that leads to the best improvement for a given step k can only be known after optimizing over $\mathcal{P}_{k+1}$. Trying all possible split decisions and optimizing all possible DSCP($\mathcal{P}_{k+1}$) would definitely require prohibitive calculation time. We present below certain techniques for making reasonable splitting decisions using a limited computing time.

3.4.1 Splitting Groups Based on a Binary Decomposition

The most general splitting approach only works with *regular* interval lengths. To determine the group to be split, a reasonable choice consists of selecting a group j^* with maximum *resource consumption spread*, *i.e.*, such that $w^{j^*}_{\max} - w^{j^*}_{\min} = \max w^j_{\max} - w^j_{\min}$. This group j^* is split into sub-groups j_1 and j_2 using a *split point* $c^* \in [w^j_{\min}..w^j_{\max}]$ that separates the elements of I^{j^*} as follows: $I^{j_1} = \{i \in I^{j^*} : \ w_i \leq c^*\}$ with $n_{j_1} = |I^{j_1}|$ and $I^{j_2} = \{i \in I^{j^*} : \ w_i > c^*\}$ with $n_{j_2} = |I^{j_2}|$. To maintain the regularity of the intervals sizes, this split point is simply determined via $c^* = \frac{t}{2^z} C^+$, where z is the minimum integer that can lead to $n_{j_1}, n_{j_2} \geq 1$ for some $t \in \mathbb{Z}_+$; the best value of t is chosen afterwords so as to minimize $|n_{j_1} - n_{j_2}|$. This basic method is general, but does not take into account any information from the current step of the optimization.

3.4.2 Group-Splitting Following an *Outside Reference Solution*

The split method from this section aims at removing some $\mathcal{P}_k$ constraints that block the optimum of DSCP(P_k) in (3.5) from advance towards an *outside reference solution* of better objective value.

Definition 5. *A solution* $\mathbf{y^u} \in \mathbb{R}^n_+$ *is an outside reference solution for* $\mathcal{P}$ *if* $\mathbf{b}^\top \mathbf{y^u} \geq OPT_{CG}$*, i.e.,* $\mathbf{y^u}$ *is associated to an upper bound of* OPT_{CG}*. This solution is either the optimal solution of* (2.2)*, or belongs to the exterior of* $\mathcal{P}$*.*

We focus on the differences between the solution that realizes $OPT(\text{DSCP}(\mathcal{P}_k))$ and a given outside reference solution $\mathbf{y^u}$. In the following, we denote by $\mathbf{y^*_k}$ the optimal solution of $\text{DSCP}(\mathcal{P}_k)$ lifted in the original space $\mathbb{R}^n_+$. The goal of this split method is to remove the constraints of $\mathcal{P}_k$ that separate $\mathbf{y^*_k}$ from $\mathbf{y^u}$, and, in the first place, those which are verified to equality by $\mathbf{y^*_k}$. In other words, we seek the constraints that should be removed to allow the solution $\mathbf{y^*_{k+1}}$ of the next iteration to be closer to $\mathbf{y^u}$ than $\mathbf{y^*_k}$. We first introduce the notion of *open direction*.

Definition 6. *Let* $\mathbf{y}$ *be a feasible solution of* $\mathcal{P}_k$ *and* $\mathbf{y^u}$ *an outside reference solution. The direction* $\mathbf{y} \to \mathbf{y^u}$ *is called "open" if there exists a small enough* $\epsilon > 0$ *such that* $\mathbf{y}_\epsilon = \mathbf{y} + \epsilon(\mathbf{y^u} - \mathbf{y}) \in \mathcal{P}$*.*

In concrete terms, there is a feasible solution $\mathbf{y}_\epsilon$ different from $\mathbf{y}$ that can be obtained by convex combination of $\mathbf{y}$ and $\mathbf{y^u}$.

We always use outside reference solutions $\mathbf{y^u}$ generated from the current $\mathbf{y^*_k}$ by the upper bounding method from Section 3.5. We will prove (Proposition 3) that this only leads to reference solutions $\mathbf{y^u}$ such that the direction $\mathbf{y^*_k} \to \mathbf{y^u}$ is open (unless if $\mathbf{b}^\top \mathbf{y^*_k} = \text{OPT}_{\text{CG}}$, in which case $\mathbf{b}^\top \mathbf{y^u} = \text{OPT}_{\text{CG}}$ and 2IDA terminates). Let us focus on the constraints of $\mathcal{P}_k$ that separate $\mathbf{y^*_k}$ from $\mathbf{y}_\epsilon = \mathbf{y^*_k} + \epsilon(\mathbf{y^u} - \mathbf{y})$ for a small enough ϵ. Since $\mathbf{y}_\epsilon$ belongs to $\mathcal{P}$, $\mathbf{y^*_k}$ and $\mathbf{y}_\epsilon$ are separated by a set of $\mathcal{P}_k$ inequalities that do not belong to $\mathcal{P}$. Such inequalities can only arise by combining valid $\mathcal{P}$ constraints with group-wise linearity restrictions associated to $\mathcal{P}_k$. We will have more to say about this process in Section 4, but for the moment is is enough to insist on the following. Dropping these non-valid restrictions could drive 2IDA towards $\mathbf{y^u}$, improving the objective value by at least some $\epsilon \mathbf{b}^\top(\mathbf{y^u} - \mathbf{y})$.

Therefore, a split decision should select a group j^* that is associated to groupwise linear restrictions separating $\mathbf{y^*_k}$ from $\mathbf{y^u}$. Since non-valid $\mathcal{P}_k$ constraints spring from enforcing the dual values in each group to follow a linear function, we consider groups whose dual values in $\mathbf{y^u}$ do not follow a linear function. Based on such information, we determine j^* based on evaluating how far $\mathbf{y^u}$ is from a linear function over each $j \in [1..k]$. An example of precise evaluation formula is provided in Section 5.3, *i.e.*, function h_3 in the cutting stock application.

Such scores can be eventually combined with other indicators based on the resource consumption spread of the groups (as in Section 3.4.1). This allows one to assign a global improvement probability score for each possible split decision. This approach does not always lead to the direction of highest improvement, but it does use information on the current primal and dual bounds to guide the search. Moreover it does not have a high computational cost when such the outside reference solution $\mathbf{y^u}$ is provided by the upper bounding approach of Section 3.5.

3.5 Upper Bounding and Open Directions based on the $\mathcal{P}_k$ Description

The method proposed in this section takes $\mathbf{y^*_k}$ (optimal solution of $\text{DSCP}(P_k)$ expressed in dimension $\mathbb{R}^n_+$) as input and returns an upper bound $\mathbf{b}^\top \mathbf{y^u}$ associated to an outside reference solution $\mathbf{y^u}$ such that:

1. if $\mathbf{b}^\top \mathbf{y^*_k} < \text{OPT}_{\text{CG}}$, the direction $\mathbf{y^*_k} \to \mathbf{y^u}$ is open
2. when $\mathbf{b}^\top \mathbf{y^*_k} = \text{OPT}_{\text{CG}}$, then $\mathbf{b}^\top \mathbf{y^u} = \text{OPT}_{\text{CG}}$.

The first property is useful for guiding the group splitting heuristic from Section 3.4.2 above. It means that $\mathcal{P}$ contains a better solution than $\mathbf{y^*_k}$ that is a linear combination of $\mathbf{y^*_k}$ and $\mathbf{y^u}$. The second property allows to stop the 2IDA iterative process as soon as $\mathbf{b}^\top \mathbf{y^*_k} = \text{OPT}_{\text{CG}}$.

3.5.1 Theoretical Modelling: the Polytope $\mathcal{P}^u$ of $\mathbf{y^*_k}$-Tight Constraints

Given the optimal solution $\mathbf{y^*_k} \in \mathbb{R}^n_+$ of $\text{OPT}(\text{DSCP}(\mathcal{P}_k))$ at current step k, we determine $\mathbf{y^u}$ by optimizing the dual set-covering problem associated to polytope $\mathcal{P}^u \supset \mathcal{P}$ defined by:

$$\mathcal{P}^u = \left\{\mathbf{y} \in \mathbb{R}^n_+ : \mathbf{a}^\top \mathbf{y} \leq 1, \forall \mathbf{a} \in \mathcal{A} \text{ such that } \mathbf{a}^\top \mathbf{y}^*_\mathbf{k} = 1\right\} \tag{3.8}$$

We say that $\mathcal{P}^u$ is the *polytope of the* $\mathbf{y}^*_\mathbf{k}$*-tight constraints* of $\mathcal{P}$. It is simply obtained from $\mathcal{P}$ by only keeping the constraints that are satisfied to equality by $\mathbf{y}^*_\mathbf{k}$. The following proposition shows that $\mathcal{P}^u \supset \mathcal{P}$, and so, $\text{OPT}(\mathcal{P}^u) \geq \text{OPT}_{\text{CG}}$, *i.e.*, $\mathbf{y^u}$ is an outside reference solution of $\mathcal{P}$ that yields the upper bound value $\mathbf{b}^\top \mathbf{y^u}$.

Proposition 3. *A solution* $\mathbf{y^u}$ *such that* $\mathbf{b}^\top \mathbf{y^u} = \max\{\mathbf{b}^\top \mathbf{y} : \ \mathbf{y} \in \mathcal{P}^u\} = OPT(DSCP(\mathcal{P}^u))$, *where* $\mathcal{P}^u$ *is defined by* (3.8)*, satisfies the following properties:*

(1) upper bounding: $\mathbf{b}^\top \mathbf{y^u} \geq OPT_{CG}$*;*

(2) open direction: *if* $\mathbf{y}^*_\mathbf{k}$ *is* ***not*** *an optimal solution of DSCP*$(\mathcal{P})$*, then* $\mathbf{b}^\top \mathbf{y}^*_\mathbf{k} < \mathbf{b}^\top \mathbf{y^u}$ *and* $\mathbf{y}^*_\mathbf{k} \rightarrow \mathbf{y^u}$ *is an open direction;*

(3) optimality proving: *if* $\mathbf{b}^\top \mathbf{y}^*_\mathbf{k} = OPT_{CG}$ *, then* $\mathbf{y^u}$ *also satisfies* $\mathbf{b}^\top \mathbf{y^u} = OPT_{CG}$*.*

Proof. The first property comes directly from the $\mathcal{P}^u$ definition (3.8). Since $\mathcal{P}^u$ is constructed from a subset of the constraints of $\mathcal{P}$, we directly have $\mathcal{P}^u \supset \mathcal{P}$. By optimizing in direction $\mathbf{b}$ over $\mathcal{P}^u$, we obtain a solution $\mathbf{y^u}$ that dominates the optimum over $\mathcal{P}$ in direction $\mathbf{b}$, *i.e.*, $\mathbf{b}^\top \mathbf{y^u} \geq \text{OPT}_{\text{CG}}$.

We now prove (2). Since $\mathbf{y}^*_\mathbf{k}$ is not an optimal solution, we obtain $\mathbf{b}^\top \mathbf{y^u} \geq \text{OPT}_{\text{CG}} > \mathbf{b}^\top \mathbf{y}^*_\mathbf{k}$. Suppose that there is no feasible convex combination of $\mathbf{y}^*_\mathbf{k}$ and $\mathbf{y^u}$ besides $\mathbf{y}^*_\mathbf{k}$. In this case, $\mathbf{y}^*_\mathbf{k}$ belongs to a facet of $\mathcal{P}$, *i.e.*, it cannot lie in the interior of $\mathcal{P}$. This face would yield a $\mathbf{y}^*_\mathbf{k}$-tight constraint that would cut off all convex combination of $\mathbf{y}^*_\mathbf{k}$ and $\mathbf{y^u}$, except $\mathbf{y}^*_\mathbf{k}$, but including $\mathbf{y^u}$. This is impossible because, by construction, $\mathbf{y^u}$ satisfies all $\mathbf{y}^*_\mathbf{k}$-tight constraints. Therefore, the segment joining $\mathbf{y}^*_\mathbf{k}$ and $\mathbf{y^u}$ does contain more solutions that belong to $\mathcal{P}$, and so, $\mathbf{y}^*_\mathbf{k} \rightarrow \mathbf{y^u}$ is an open direction.

The last property (3) comes from the construction of $\mathcal{P}^u$. If $\mathbf{y}^*_\mathbf{k}$ is optimal in $\mathcal{P}$, its basic constraints need to be $\mathbf{y}^*_\mathbf{k}$-tight, and so, they belong to $\mathcal{P}^u$. These constraints are sufficient to ensure the optimality of $\mathbf{y^u}$ in $\mathcal{P}$. □

3.5.2 Practical Aspects: Constructing $\mathcal{P}^u$

We solve $\text{DSCP}(\mathcal{P}^u)$ by column generation. $\text{DSCP}(\mathcal{P}^u)$ has exactly the same structure as $\text{DSCP}(\mathcal{P})$ in (2.2), but the set of possible constraints is restricted to those that are $\mathbf{y}^*_\mathbf{k}$-tight. The pricing sub-problem requires finding a configuration $\mathbf{a} \in \mathcal{A}$ with $\mathbf{a}^\top \mathbf{y}^*_\mathbf{k} = 1$ ($\mathbf{y}^*_\mathbf{k}$-tightness) that maximizes $\mathbf{y}^\top \mathbf{a}$ for the current dual solution $\mathbf{y}$. This pricing sub-problem can be solved exactly as the initial pricing sub-problem (Section 2.1), by replacing $\max_{\mathbf{a} \in \mathcal{A}} \mathbf{y}^\top \mathbf{a}$ in (2.3) with a hierarchical objective function $\max_{\mathbf{a} \in \mathcal{A}} (M\mathbf{y}^*_\mathbf{k} + \mathbf{y})^\top \mathbf{a}$, where M is a sufficiently large value.

In practice, this behavior is obtained using a lexicographic maximization objective, *i.e.*, first maximize $\mathbf{a}^\top \mathbf{y}^*_\mathbf{k}$, and, subject to this, maximize $\mathbf{a}^\top \mathbf{y}$. The original dynamic programming scheme (Section 2.1) is not modified and the number of states is the same. Instead of computing a unique profit function $P_{\max}$ in the recursion (2.4), the *lexicographic dynamic program* computes an ordered pair of profit functions $(P^*_{\max}, P^u_{\max})$. For each state $(c, i) \in [0, C^+] \times [1..n]$, $P^*_{\max}(c, i)$ and $P^u_{\max}(c, i)$ represent the maximum of $(\mathbf{y}^*_\mathbf{k})^\top \mathbf{a}$ and, respectively, of $\mathbf{y}^\top \mathbf{a}$ over all (partial) configurations $\mathbf{a}$ in state (c, i), *i.e.*, consuming a resource amount of c only using elements of $[1..i]$. The recursion (2.4) evolves to a lexicographic maximization formula.

Proposition 4. *Given a dual solution* $\mathbf{y}$*, the above lexicographic dynamic program (optimizing* $({\mathbf{y}^*_\mathbf{k}}^\top \mathbf{a}, \mathbf{y}^\top \mathbf{a})$ *lexicographically) determines a configuration* $\mathbf{a} \in \mathcal{A}$ *that maximizes* $\mathbf{y}^\top \mathbf{a}$ *subject to* $(\mathbf{y}^*_\mathbf{k})^\top \mathbf{a} = 1$*.*

Proof. The evolution of the lexicographic dynamic program can be interpreted as follows: at each recursion step, the lexicographic maximization formula actually maximizes $(\mathbf{y}^*_\mathbf{k})^\top \mathbf{a}$, breaking ties using the value of $\mathbf{y}^\top \mathbf{a}$. As such, the $\mathbf{y}$ objective is only used for tie breaking, and so, the lexicographic dynamic program does return a configuration $\mathbf{a}^*$ that maximizes the $\mathbf{y}^*_\mathbf{k}$ objective.

As such, $(\mathbf{y}^*_\mathbf{k})^\top \mathbf{a}^*$ achieves the maximum value of 1, because at least some "≤ 1" constraints in (3.5) need to be saturated by $\mathbf{y}^*_\mathbf{k}$ (otherwise, no other (3.5) constraint can render infeasible a higher value solution such as $\mathbf{y}^*_\mathbf{k} + \epsilon \mathbf{1}_n$, with a sufficiently small ϵ). □

maybe insist that any $\mathbf{y}^*_\mathbf{k}$ belongs to a $\mathcal{P}$ facet. It is easy to check.

Consequently, the complexity of the DP pricing for DSCP($\mathcal{P}^u$) is the same as the complexity of the DP pricing for DSCP($\mathcal{P}$). However, since $\mathcal{P}^u$ is defined by a far smaller number of constraints than $\mathcal{P}$, the proposed upper bounding method is generally faster.

3.6 $\mathcal{P}_{k+1}$ Optimization: Using $\mathcal{P}_k$ to Warm-Start the Column Generation

After solving DSCP($\mathcal{P}_k$) to optimality at iteration k, 2IDA splits a group to obtain a new partition G_{k+1} with $k+1$ groups. A new program (3.5), associated with a new polytope $\mathcal{P}_{k+1}$ has to be optimized. One could directly apply from scratch the column generation method from Section 3.2.2 after each k incrementation, but this may globally require a too large computational cost. Therefore, we use the columns already generated at iteration k to *warm-start* the column generation phase at the next iteration $k+1$. The main goal is to perform an incremental iterative description of the polytopes $\mathcal{P}_1, \mathcal{P}_2, \ldots$.

Let j^* be the index of the group that is split after iteration k, and j_1, j_2 be the two new (sub-)group indices. Right after taking the split decision, 2IDA (see Algorithm 1) starts building $\mathcal{P}_{k+1}$. It first determines the new $\mathcal{P}_{k+1}$ coefficients ($N^-(j_1, c^{j_1})$, $N^+(j_1, c^{j_1})$, etc.) by calling `detPkCoefs` (Line 8). This routine applies on groups j_1 and j_2 the coefficient calculation preprocessing stage from Section 3.2.1. In fact, the coefficients of j_1 can be directly recovered from the program that was run on j^*, supposing that the dynamic program for I^{j^*} considered the I^{j_1} elements before I^{j_2}.

Right after splitting j^*, the next step of 2IDA (Line 9) generates some $\mathcal{P}_{k+1}$ constraints by lifting existing $\mathcal{P}_k$ constraints from dimension $\mathbb{R}^{2k}$ to dimension $\mathbb{R}^{2k+2}$. The optimal solution of DSCP($\mathcal{P}_k$) is also lifted to construct an initial feasible solution for $\mathcal{P}_{k+1}$, *i.e.*, 2IDA keeps this solution unchanged over groups of $[1..k] \setminus \{j^*\}$, and applies $\alpha^{j_1} = \alpha^{j_2} = \alpha^{j^*}$ and $\beta^{j_1} = \beta^{j_2} = \beta^{j^*}$ for j_1 and j_2. Starting from this $\mathcal{P}_{k+1}$ solution, 2IDA first applies column generation to optimize over an intermediate polytope (noted $\mathcal{P}'_{k+1}$, see below) containing inherited constraints only. If the optimal solution of DSCP($\mathcal{P}_k$) remains optimal for DSCP(P'_{k+1}), there is no need to continue optimizing over $\mathcal{P}_{k+1}$. Otherwise, when no more inherited constraints lead to negative reduced cost columns, 2IDA proceeds to the full CG from Section 3.2.2, *i.e.*, when Algorithm 1 goes on to Line 4.

More formally, we note $\mathcal{A}^{\text{base}}_k$ the set of configurations associated to constraints of $\mathcal{P}_k$ that are either related to basic primal variables, or non-basic primal variables of reduced cost 0 (with regards to $\mathbf{y}^*_\mathbf{k}$). We consider only the constraints that have been explicitly generated when optimizing $\mathcal{P}_k$ in (3.5). Our approach is equivalent to first constructing a dual polytope $\mathcal{P}'_{k+1}$ focusing on a set of *inherited constraints*: $\mathcal{A}'_{k+1} = \left\{\mathbf{a} \in \mathcal{A}_{k+1} : \exists \mathbf{a}' \in \mathcal{A}^{\text{base}}_k, \text{ such that } c^j_a = c^j_{a'} \text{ and } N^j_a = N^j_{a'}\ \forall j \in [1..k] \setminus \{j^*\}\right\}$, where $\mathcal{A}_{k+1}$ is the set of (non-dominated) $\mathcal{P}_{k+1}$ constraints in (3.5). Since $\mathcal{A}'_{k+1} \subset \mathcal{A}_{k+1}$, we have $\text{OPT}(\text{DSCP}(\mathcal{P}'_{k+1}))) \geq \text{OPT}(\text{DSCP}(\mathcal{P}_{k+1})))$.

Except for the split group j^*, all coefficients of existing $\mathcal{A}^{\text{base}}_k$ configurations can be transmitted to recover more rapidly some $\mathcal{A}'_{k+1}$ configurations. Each time the pricing sub-problem is called, we seek and add negative reduced cost columns in two phases: (i) try to find columns associated to configurations in $\mathcal{A}'_{k+1}$ and (ii) search column using the original method from Section 3.2.2. For each configuration in $\mathcal{A}^{\text{base}}_k$, step (i) seeks a configuration of negative reduced cost that uses the same coefficients (c^j, N^j) for any group $j \neq j^*$ and new coefficients for j_1 and j_2. This is solved using the multi-choice dynamic program from Section 3.2.2, but with only two decision levels j_1 and j_2.

This latter dynamic programming is significantly faster than the original one from Section 3.2.2. Besides only needing two levels instead of k, the maximum resource amount to consider is also reduced from C^+ to $C^+ - \sum_{j \in [1..k]-\{j^*\}} c^j$ (the space occupied by inherited values). In certain cases, $\text{OPT}(\text{DSCP}(\mathcal{P}'_{k+1})) = \text{OPT}(\text{DSCP}(\mathcal{P}_k))$ and this is enough to conclude $\text{OPT}(\text{DSCP}(\mathcal{P}_{k+1})) = \text{OPT}(\text{DSCP}(\mathcal{P}_k))$, because the 2IDA bounds can not decrease by incrementing k. In practical terms, this fully justifies using the aggregated pricing approach from Section 3.2.2: when using this inheritance with two levels, it can sometimes incrementally compute lb_{k+1} from lb_k in virtually no-time. The full algorithmic template is presented in Algorithm 2: Phase 1 roughly corresponds to Line 9 (in Algorithm 1) and Phase 2 to Line 4 (with incremented k).

Algorithm 2: Two-Phase CG for optimizing $\text{DSCP}(\mathcal{P}_{k+1})$: construct $\mathcal{P}'_{k+1}$ then $\mathcal{P}_{k+1}$

Data: Optimal $\text{DSCP}(\mathcal{P}_k)$ solution $[\alpha^1 \dots \alpha^k \ \beta^1 \dots \beta^k]^\top$, existing constraints $\mathcal{A}_k^{\text{base}}$
Result: $\text{OPT}(\text{DSCP}(\mathcal{P}_{k+1}))$

Phase 1: Lift current solution and describe $\mathcal{P}'_{k+1}$
Express current $[\alpha^1 \dots \alpha^k \ \beta^1 \dots \beta^k]^\top$ as a feasible $\text{DSCP}(\mathcal{P}_{k+1})$ solution:
- Keep the values α^j and β^j for all $j \neq j^*$ // *no change outside split group j^**
- $\alpha^{j_1} \leftarrow \alpha^j$, $\alpha^{j_2} \leftarrow \alpha^j$
- $\beta^{j_1} \leftarrow \beta^j$, $\beta^{j_2} \leftarrow \beta^j$

repeat
  for $\mathbf{a} \in \mathcal{A}_k^{base}$ **do**
    – solve the aggregated multiple-choice pricing variant (Section 3.2.2) with 2 elements (j_1 and j_2) and capacity limits $C^- - \sum_{j \in [1..k]-\{j^*\}} c_a^j$ and $C^+ - \sum_{j \in [1..k]-\{j^*\}} c_a^j$ to obtain a configuration $\mathbf{a}'$
    if $\mathbf{a}'$ *has a negative reduced cost* **then**
      $\mathcal{A}'_{k+1} \leftarrow \mathcal{A}'_{k+1} \cup \{\mathbf{a}'\}$
    end
  end
  – solve the Reduced Master Problem using only configurations $\mathcal{A}'_{k+1}$
  – update $\text{OPT}(\text{DSCP}(P'_{k+1}))$ and the current dual solution $[\alpha, \beta]$;
until *no configuration* $\mathbf{a}'$ *of negative reduced can be found*

Phase 2: Describe $\mathcal{P}_{k+1}$
if $OPT(DSCP(\mathcal{P}'_{k+1})) = OPT(DSCP(P_k))$ **then**
  return $\text{OPT}(\text{DSCP}(P_k))$ // $OPT(DSCP(P_k)) = OPT(DSCP(P_{k+1}))$
end
repeat
  – solve the aggregated multiple-choice pricing variant (Section 3.2.2) on $k+1$ levels to obtain a new configuration $\mathbf{a}$
  if $\mathbf{a}$ *has a negative reduced cost* **then**
    $\mathcal{A}_{k+1} \leftarrow \mathcal{A}_{k+1} \cup \{\mathbf{a}\}$
    – solve the RMP related to configurations $\mathcal{A}_{k+1}$ and update the current dual values
  end
until *no configuration* $\mathbf{a}$ *of negative reduced can be found*
return $\text{OPT}(\text{DSCP}(\mathcal{P}_{k+1}))$

4 Comparing Linear and Equality Aggregations

We now investigate how group-wise linear restrictions mix with legitimate $\mathcal{P}$ constraints to produce the "inner polytope" $\mathcal{P}_k \subset \mathcal{P}$. We motivate on the use of linear aggregation with regard to simpler equality aggregations (*e.g.*, $y_i = y_j$, for aggregated $i, j \in [1..n]$). Equality aggregations are mostly

useful when one can assume that elements of similar weights will be associated to the same dual value in an optimal solution. This is sometimes true, for instance, in certain instances of the cutting-stock problem, if the weights of the aggregated items are similar. More generally, it is reasonable to perform equality aggregations on rather similar entities in the model, such as, related scheduling tasks [10, 11, 3], close time instants [19], nearby locations, etc. However, in capacitated problems, elements with significantly different resource consumptions can hardly be considered equivalent, as the capacity constraints can substantially change the column sets covering such elements.

2IDA could simply be transformed to perform equality aggregation by imposing $\alpha^j = 0, \forall j \in [1..k]$. However, for both linearity and equality restrictions, the aggregated pricing for $\mathcal{P}_k$ generates only constraints that are valid for P, yet $\mathcal{P}$ contains some solutions that cannot be projected into feasible $\mathcal{P}_k$ solutions. Non-valid (overly-strong) $\mathcal{P}$ constraints implicitly arise during the $\mathcal{P}_k$ construction by combining valid columns and *exchange vectors* [27] that are *not* valid for $\mathcal{P}$.

Let us illustrate the process on a valid $\mathcal{P}$ constraint involving some terms $a_i y_i + a_j y_j$, *i.e.*, the corresponding column has the form $[a_i \dots a_j \dots]^\top$ (assume without generality loss that i is the first element), where the dotted elements indicates other values of the column. An equality aggregation $y_i = y_j$ would makes a_i and a_j interchangeable in the above column. This property generates exchange vectors of the form $[1 \dots -1 \dots]^\top$, where the dotted elements represent 0s. A valid constraint and an exchange vector produce implicit constraints as follows:

$$\begin{bmatrix} a_i \\ \vdots \\ a_j \\ \vdots \end{bmatrix} + \psi \begin{bmatrix} 1 \\ \vdots \\ -1 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} a_i+\psi \\ \vdots \\ a_j-\psi \\ \vdots \end{bmatrix}, \tag{4.1}$$

where $\psi \in [-a_i, a_j]$. The obtained constraint may be non feasible. By setting ψ to $-a_i$ or a_j, one respectively replaces a_i by a_j, or a_j by a_i. Since we deal with equality restrictions, such replacements are always bidirectional (negative or positive). Using equality aggregations, the exchange vector might relax the maximum (resp. minimum) resource constraint if it replaces an element by an element of larger (resp. smaller) resource consumption.

We will now see that the dual linearity restrictions produce exchange vectors that are fractional. The constraints of (3.1) (see p. 5) can be written using the $\mathbf{y}$ variables only. Using Definition 1 (p. 6), we can write $y^j_{\min} = \alpha w^j_{\min} + \beta$, and $y^j_{\max} = \alpha w^j_{\max} + \beta$, where $y^j_{\max}$ and $y^j_{\min}$ correspond to the elements of I^j of largest (respectively smallest) consumption. This gives us $\alpha = \frac{y^j_{\max} - y^j_{\min}}{w^j_{\max} - w^j_{\min}}$ and $\beta = \frac{w^j_{\min} y^j_{\max} - w^j_{\max} y^j_{\min}}{w^j_{\min} - w^j_{\max}}$. Then, for each element i of group j different from these two extrema, the linearity constraint is equivalent to add a constraint $y_i = \frac{y^j_{\max} - y^j_{\min}}{w^j_{\max} - w^j_{\min}} w_i + \frac{w^j_{\min} y^j_{\max} - w^j_{\max} y^j_{\min}}{w^j_{\min} - w^j_{\max}}$.

Without restricting generality, we consider that the first element of I^j corresponds to $y^j_{\min}$ and the last one to $y^j_{\max}$. Therefore, the exchange vectors produced restricted to group j have the following form:

$$\begin{bmatrix} a^j_{\min} \\ \vdots \\ a_i \\ \vdots \\ a^j_{\max} \end{bmatrix} + \psi \begin{bmatrix} \frac{w^j_{\max} - w_i}{w^j_{\max} - w^j_{\min}} \\ \vdots \\ -1 \\ \vdots \\ \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}} \end{bmatrix} \rightarrow \begin{bmatrix} a^j_{\min} + \psi \frac{w^j_{\max} - w_i}{w^j_{\max} - w^j_{\min}} \\ \vdots \\ a_i - \psi \\ \vdots \\ a^j_{\max} + \psi \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}} \end{bmatrix}, \tag{4.2}$$

where ψ can be positive or negative, which may respectively decrease or increase the coefficient of a_i.

An implicit configuration $\hat{\mathbf{a}}$ generated this way can have fractional coefficients. Therefore, it may lead to violated patterns because it relaxes the integrity of the variables, but it cannot violate the resource constraints—*i.e.*, $C^- \leq \hat{\mathbf{a}}^\top \mathbf{w} \leq C^+$ still holds. We now show this result formally.

Proposition 5. *Non-valid configurations* $\hat{\mathbf{a}}$ *produced by linearity aggregation from a feasible configuration* $\mathbf{a}$ *have the same resource consumption as* $\mathbf{a}$*. Such non-valid configurations cannot violate resource constraints. This is generally false for configurations produced by equality aggregations.*

Proof. We first show that the equality aggregation may lead to non-valid configurations $\hat{\mathbf{a}}$ that violate the resource constraint. It is sufficient to exhibit an example. Take $C^- = 8$ and $C^+ = 10$, and two elements of resource consumption $w_1 = 8$ and $w_2 = 3$ in the same group. Configuration $[0, 3]^\top$ is valid. A exchange vector $[1, -1]^\top$ with $\psi = 3$ leads via (4.1) to $[3, 0]^\top$, which corresponds to a total resource consumption of 24. For the minimum resource consumption, an example can be produced by taking the valid configuration $[1, 0]^\top$ and using $\psi = -1$ in (4.1).

We now show that non-valid configurations $\hat{\mathbf{a}}$ generated by linearity restrictions do not violate the resource constraint because the total resource consumption cannot be modified by linearity-based exchange vectors. Recall that the configurations $\mathbf{a}$ generated by the pricing sub-problem are valid configurations, *i.e.* $C^- \leq \mathbf{a}^\top \mathbf{w} \leq C^+$. An implicit configuration $\hat{\mathbf{a}}$ is obtained by iteratively applying exchange vectors to a valid configuration $\mathbf{a}$.

Let $\mathbf{a} = [a^j_{\min} \dots a_i \dots a^j_{\max}]^\top$ be a feasible configuration and $\mathbf{e}$ the exchange vector $\left[\frac{w^j_{\max} - w_i}{w^j_{\max} - w^j_{\min}} \dots, -1, \dots \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}}\right]^\top$. We observe in (4.2) that combining $\mathbf{a}$ and $\mathbf{e}$ can lead to new artificial configurations the form $\mathbf{a} + \psi\mathbf{e}$.

Initially the resource consumption of configuration $\mathbf{a}$ is $\mathbf{a}^\top \mathbf{w}$. Note that in $\mathbf{a} + \psi\mathbf{e}$, the only coefficients to be modified are related to $a^j_{\min}$, $a^j_{\max}$ and a_i. In the original configuration, the total resource consumption of these three elements is $a^j_{\min} w^j_{\min} + a_i w_i + a^j_{\max} w^j_{\max}$. In $\mathbf{a} + \psi\mathbf{e}$, this resource consumption becomes:

$$(a^j_{\min} + \frac{w^j_{\max} - w_i}{w^j_{\max} - w^j_{\min}}\psi) w^j_{\min} + (a_i - \psi) w_i + (a^j_{\max} + \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}}\psi) w^j_{\max},$$

which is equivalent to:

$$a^j_{\min} w^j_{\min} + a_i w_i + a^j_{\max} w^j_{\max} + \frac{w^j_{\max} - w_i}{w^j_{\max} - w^j_{\min}}\psi w^j_{\min} - \frac{\psi w_i (w^j_{\max} - w^j_{\min})}{w^j_{\max} - w^j_{\min}} + \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}}\psi w^j_{\max}$$

This simplifies to $a^j_{\min} w^j_{\min} + a_i w_i + a^j_{\max} w^j_{\max}$, which means that the resource consumption is the same in $\mathbf{a}$ and $\mathbf{a} + \psi\mathbf{e}$. By iteratively adding exchange vectors to the initial valid configuration $\mathbf{a}$, a configuration $\hat{\mathbf{a}}$ such that $C^- \leq \hat{\mathbf{a}}^\top \mathbf{w} = \mathbf{a}^\top \mathbf{w} \leq C^+$ is obtained. Therefore, any such configuration $\hat{\mathbf{a}}$ does not violate the resource constraint.

□

5 Cutting-Stock Application and Implementation Aspects

In order to validate our methodology, we applied 2IDA to the well-known Cutting-Stock Problem. We first discuss the effectiveness of linearity restrictions compared to equality restrictions in this special case. Then, we describe in greater detail some group-splitting strategies improved by additional problem knowledge.

The cutting-stock problem is the following. We are given a bin size $C \in \mathbb{N}$ and a list $1, \dots, n$ of items. Each item i has a weight w_i and a demand b_i. The objective is to minimize the number of bins to use to accommodate all items. This problem can be solved [13] by an extended formulation whose master problem is a set-covering model, and whose pricing sub-problem is a knapsack problem. Therefore, it lies in the scope of our study, fitting the initial model (2.2) very well. We use weights of the items for ressource consumption and the total capacity consumption must be between 0 and C^+. To simplify the notation, we will simply note C^+ as C.

5.1 Linear Restrictions vs. Equality Restrictions

Since the equality aggregation is a special case of linear aggregation, the quality of the bound obtained by the latter will be better. For $k = 1$, the linear aggregation even has a worst-case performance that cannot be reached by the equality aggregation.

Proposition 6. *The asymptotic worst case performance of 2IDA with $k = 1$ for the standard Cutting-Stock Problem is $1/2$, and this ratio is tight.*

Proof. Polytope $\mathcal{P}_1$ contains the solution $y_i = \frac{w_i}{C}$. This dual solution is equivalent to the optimal solution of the model of [15], which is known to have an asymptotic worst-case performance of $1/2$.

To prove the tightness, consider an instance with $n = 3$, $\mathbf{w} = \left[\epsilon,\ \frac{C}{2} + \epsilon,\ C - \epsilon\right]$ and $\mathbf{b} = [1,\ M,\ 1]$, where $\epsilon > 0$ is a sufficiently small real and M is a sufficiently large integer. The optimal solution of this instance is $M + 1$. We observe that the set of R^1 of feasible resource consumptions is $\{\epsilon, \frac{C}{2} + \epsilon, \frac{C}{2} + 2\epsilon, C - \epsilon, C\}$. Since $w_2 = \frac{C}{2} + \epsilon$ can be as close to $\frac{w_1 + w_3}{2}$ as desired (using a small enough ϵ), y_2 can also be arbitrarily close to $\frac{y_1 + y_3}{2}$ – as long as we have only one group, *i.e.*, $k = 1$. We now observe that $y_1 + y_3 \leq 1$, based on the pattern with total consumption C. As such, we can conclude that y_2 can be arbitrarily close to $\frac{1}{2}$; as such, the objective value $\mathrm{OPT}(\mathrm{DSCP}(\mathcal{P}_1))$ is $\mathbf{b}^\top \mathbf{y} = b_1 y_1 + b_2 y_2 + b_3 y_3 = y_1 + M y_2 + y_3$, which has an asymptotic value (when $\epsilon \to 0$) of $\frac{M}{2} + 1$, while the instance has an optimum of $M + 1$. □

This result does not hold for the equality aggregation. For $k = 1$, all dual values are equal to $\frac{1}{\lfloor C / w_{\min} \rfloor}$, otherwise the valid dual-constraint that uses $\lfloor C / w_{\min} \rfloor$ items of size $w_{\min}$ would be excluded. Therefore, the ratio between the value found by the model and the optimal solution is not bounded by any constant.

5.2 Split Selection Based on Dual-Feasible Functions

In Section 3.4, we have mentioned general principles for making splitting decisions in 2IDA. For the Cutting-Stock Problem, we can also use ad-hoc methods based on the knowledge of good dual solutions (obtained by applying so-called Dual-Feasible Functions [18, 7]).

Definition 7. *A function $f : [0, C] \to [0, 1]$ is a dual-feasible function (DFF), if the following holds $\sum_{i \in I} b_i w_i \leq C \implies \sum_{i \in I} b_i f(w_i) \leq 1$ for any index set I such that $b_i \in \mathbb{Z}^+$ and $w_i > 0$.*

To obtain a valid lower bound for the cutting-stock problem, it is sufficient to apply a dual-feasible function on the weights of the items and to sum up the obtained values. This way, a dual feasible function generate a valid dual solution $\mathbf{y}$: it takes the weights as an input, and associate to each weight w_i the corresponding dual value y_i.

Given a good dual-feasible solution f, a good method for splitting groups consists of splitting $[0, C]$ in such a way that f is linear over each the groups. Therefore, our group split method dedicated to the cutting-stock problem tries to reproduce the structure of the best dual-feasible function available. The motivation comes from the fact that most of the classical DFFs have a piece-wise linear form. The different pieces define a natural partition of the element set. By identifying all intervals of $[0, C]$ corresponding with the different pieces of the DFF, our DFF-based split method computes a priori k intervals before starting constructing $\mathcal{P}_k$. If the function has k pieces, splitting the dual values into k groups leads 2IDA at least to the same feasible solutions as those returned by the DFFs. However, the best function is not always useful. For example, $f(x) = \frac{x}{C}$ only defines $k = 1$ interval. Therefore, such specific functions are not used in practice.

5.3 Split Selection Based on Outside Reference Solutions

We now introduce a split method that implements the principles from Section 3.4.2 and tunes the evaluation for Cutting-Stock. While many details below are especially tailored for Cutting-Stock, similar formulas can be used for other applications.

Since elements are sorted by increasing weight within each group, we can express a split decision by a pair (j, n_{j_1}), where j is the group to be split, and n_{j_1} the index of the highest element in the first subgroup j_1. Our splitting method assigns scores $h(j, n_{j_1})$ to all potential splits. Three types of information are used to determine a total score $h(j, n_{j_1}) = h_1(j, n_{j_1}) \cdot h_2(j, n_{j_1}) \cdot h_3(j, n_{j_1})$. In the end, the potential split of maximal score is selected.

The first two scores h_1 and h_2 are based on the simple ad-hoc considerations related to the cutting-stock problem. Score $h_1(j, n_{j_i})$ is based on weight spread information, *i.e.*, $h_1(j, n_{j_1}) = w^j_{\max} - w^j_{\min}$ if $1 < n_{j_1}, n_j - n_{j_1}$ or 1 otherwise. The special cases of splitting a group at positions 1 or $n_j - 1$ should be avoided, since they produce a subgroup of size 1. However, in general, the probability of choosing a group j is proportional to its weight spread, as it is reasonable to split larger segments first.

Score h_2 evaluates the impact of a group based on its relative position in $[1..k]$. The "extremal" groups $j = 1$ and $j = k$ are always associated to the lightest and respectively the heaviest elements and therefore are critical in the case of cutting-stock. These two groups are given a bonus by scoring function h_2. This part of the procedure is clearly hand-tailored for the cutting-stock and is not likely to generalize to other problems.

The major part of the score is given by h_3. It is based on a comparison of the current optimal solution $\mathbf{y^*_k}$ of $\mathcal{P}_k$ to an outside reference solution such that $\mathbf{y^*_k} \to \mathbf{y^u}$ is an open direction. As hinted in Section 3.4.2, we assign higher scores to groups that are more likely to lead (using the process from Section 4) to implicit non-valid constraints that separate $\mathbf{y^*_k}$ from $\mathbf{y^u}$. For this, we compare $\mathbf{y^*_k}$ with $\mathbf{y^u}$. While $\mathbf{y^*_k}$ is group-wise linear structure, this is not the case for $\mathbf{y^u}$; we actually try to find groups on which the difference $\mathbf{y^u} - \mathbf{y^*_k}$ is far from indicating a linear transformation.

The difference between $\mathbf{y^*_k}$ and $\mathbf{y^u}$ for a given group j is evaluated using the Δ operator, defined by

$$\Delta^j_{i_1,i_2}(\mathbf{y^*_k}, \mathbf{y^u}) = \sum_{i=i_1}^{i_2} b^j_i \cdot (\mathbf{y^*_k} - \mathbf{y^u})^j_i, \tag{5.1}$$

where $(\mathbf{y^*_k} - \mathbf{y^u})^j_i$ is the i^{th} position of the vector $\mathbf{y^*_k} - \mathbf{y^u}$ restricted to the elements of group j. To simplify the notations, we will hereafter ignore the argument $(\mathbf{y^*_k}, \mathbf{y^u})$ of the operator $\Delta^j_{i_1,i_2}$; no confusion can arise because all comparisons below concern $\mathbf{y^*_k}$ and $\mathbf{y^u}$.

Operator Δ is useful for detecting differences between $\mathbf{y^*_k}$ and $\mathbf{y^u}$ and proposing useful splits. Let us analyze the case where the global Δ evaluation over group j is positive—*i.e.*, consider $\Delta^j_{1,n_j} > 0$ (the opposite case $\Delta^j_{1,n_j} \leq 0$ is symmetric). This indicates a global positive "trend" on group j: the objective value of $\mathbf{y^*_k}$ could be improved by (non-linearly) increasing most dual values in group j; such operations can be performed while maintaining feasibility in $\mathcal{P}$. Secondly, if $\Delta^j_{1,i}$ is negative for some $i \in [1..n_j)$, a group split at (j, i) would allow 2IDA to decrease the first i dual values and increase the others. In this manner, 2IDA would follow the open direction $\mathbf{y^*_k} \to \mathbf{y^u}$. A more frequent situation is associated to a segment $[i_a, i_b] \in [1..n_j]$ such that $\Delta^j_{i_a,i_b} < 0$; to isolate dual values of $[i_a..i_b]$ from the rest of the group, one should consider split decisions $(j, i_a - 1)$ or $j(j, i_b)$.

For this, we divide $[1..n_j]$ in three segments $[1..i_a)$, $[i_a..i_b]$, $(i_b..n_j]$, determining i_a and i_b as follows: $i_a = \max\{i \in [1..n_j + 1] :\ \Delta^j_{1,i-1} \geq 0\}$ and $i_b = \min\{i \in [0..n_j] :\ \Delta^j_{i+1,n_j} \geq 0\}$. Let us first consider $i_a \leq i_b$. The middle segment $[i_a, i_b]$ has an "inverse trend" compared to $[1..n_j]$, *i.e.*, $\mathbf{y^*_k}$ could feasibly improve in $\mathcal{P}$, by decreasing its values over $[i_a, i_b]$, while increasing its values over the rest of $[1..n_j]$. Roughly speaking, we are looking for a sequence of indices $[i_a, ..i_b]$ on which $\mathbf{y^*_k}$ could decrease by dropping constraints that link elements of $[i_a..i_b]$ and other elements from the group. The most fortunate case is related to $i_a = 1$ (or, analogously, $i_b = n_j$), because this would make $[1..i_a)$ empty; as such, a split at i_b would surely allow the first i_a elements to decrease while increasing the others. The scoring function h^3 is determined by: $h^3(j, i_a) = \Delta^j_{1,i_a-1}$, $h^3(j, i_b) = \Delta^j_{i_b+1,n_j}$ and $h^3(j, i) = 0, \forall i \notin \{i_a, i_b\}$. Finally, if the above formulas lead to $i_a > i_b$, we simply consider that the direction $\mathbf{y^*_k} \to \mathbf{y^u}$ does not give any useful information, and we set $h^3(j, i) = 1,\ \forall i \in [1..n_j]$.

6 Numerical Evaluation

We first evaluate 2IDA on the Cutting Stock Problem (CSP) in Sections 6.2-6.4. Section 6.5 continues the evaluation with experiments on the Cutting Stock Problem with Maximum Waste (CSP-MW).

We first focus on investigating the internal 2IDA dynamics (Section 6.2), *i.e.*, we report the running time and the number of iterations (sub-problems) for the lower and the upper bounding calculations. In Section 6.3, we compare the intermediate 2IDA lower bounds with the classical Lagrangian bounds in column generation. Section 6.4 validates the fact that, in many cases, 2IDA reaches the final CG optimum (OPT_{CG}) more rapidly than classical Column Generation (CG).

6.1 Experimental Conditions and CSP Instances

We implemented both 2IDA and a classical non-stabilized CG algorithm based on Dynamic Programming (DP) for pricing the columns. A similar implementation approach (using lists of states) has been used for all DP routines of 2IDA, namely: the coefficient calculation preprocessing stage (Section 3.2.1), the multiple-choice aggregated pricing for optimizing DSCP($\mathcal{P}_k$) in Section 3.2.2 (or in "constraint inheritance" process from Algorithm 2), the upper bounding pricing for optimizing over $\mathcal{P}^u$ (Section 3.5.2).

The first split decisions are made based on the intervals returned by some classical dual-feasible function (DFF, Section 5.2). We only used DFFs with relatively few intervals (maximum 15). As soon as 2IDA requires more intervals, the upper bound and the corresponding outside reference solution (when available) are used to guide further group split decisions (see Sections 3.4.2 or 5.3). We turn to classical CG if $\lceil \text{lb}_k \rceil + 1 = \lceil \text{ub}_k \rceil$ or if $k = 10$. More exactly, when 10 iterations are not sufficient to reach optimality, the current set of columns used for upper bounding is used to start a CG phase that is run up to optimality.

We used a large set of instances from a dozen of benchmark sets (3127 individual instances). Their characteristics are described in great detail in Appendix A. The times we report are obtained using the same computing setting. All technical characteristics (the CPU model, LP solver, source code, etc.) are also specified in Appendix A.

6.2 2IDA Profile: Computing Effort and Usefulness of Upper Bounding

Computing an upper bound is not a mandatory part in 2IDA. Since it requires running non-aggregated column generation, it is interesting to evaluate the impact of the upper bound in term of total computing time.

Table 1 presents the computing effort in terms of CPU time and sub-problems needed, both for lower bound and intermediate upper bounding. We actually compare two 2IDA versions: one using the intermediate upper bounds from Section 3.5 ("yes" in Column 2) and a "pure" 2IDA version that does not use such intermediate upper bounds ("no" in Column 2). However, recall (Section 6.1) that we always turn to the classical CG algorithm if the optimum is not reached at $k = 10$. As such, the indicated upper bounding computing effort includes both the intermediate upper bounds and the upper bounds computed during the final CG process.

The first conclusion is that 2IDA is generally more efficient when intermediate upper bounds are used. These bounds are useful to guide the split decision or to stop the process more rapidly. The number of $\mathbf{y}^*_\mathbf{k}$-*tight constraints* in the polytope $\mathcal{P}^u$ (see Section 3.5) is significantly smaller than the total number of constraints in $\mathcal{P}$. Therefore, the optimization of $\mathcal{P}^u$ does not introduce prohibitively large slowdowns. As expected, the efficiency of 2IDA is greatly improved when the optimality can be proved only using intermediate lower bounds for some $k < 10$.

For full convergence, 2IDA spends more computing time on upper bounding than on lower bounding. The fact that the method switches to classical CG after a number of iterations increases the difference between the UB and LB computing efforts. This difference can be significant, most notably for `vb50-3` and `vb50-5`, where the lower bounding stops at $k = 1$ and classical CG is

Instance set	Intermediate upper bounds	Avg. CPU			LB pricing calls			UB pricing calls		
		Total	LB	UB	min	avg	max	min	avg	max
vb10	yes	1442	265	1178	11	28	60	10	12	18
	no	342	342	0	13	36	104	0	0	0
vb20	yes	9161	3162	5999	0	58	158	22	31	59
	no	11958	8029	3929	22	87	150	19	29	49
vb50-1	yes	39827	18620	21207	5	72	120	50	107	212
	no	33719	16556	17163	14	75	129	52	107	149
vb50-2	yes	89432	31087	58345	0	56	133	74	150	210
	no	75928	37987	37941	45	81	196	95	139	188
vb50-3	yes	44068	3745	40323	0	0	0	60	80	94
	no	100279	59671	40608	12	26	57	60	80	94
vb50-4	yes	66498	28142	38356	56	106	148	109	136	179
	no	43011	24907	18104	68	113	165	98	127	167
vb50-5	yes	28490	2910	25580	0	0	0	58	62	72
	no	58700	33095	25606	16	30	45	58	62	72
vbInd	yes	7580	670	6910	0	15	65	4	28	89
	no	3800	1883	1917	5	41	74	0	16	70
m01	yes	818	84	734	0	6	51	55	157	321
	no	1394	628	766	14	27	74	72	149	317
m20	yes	423	135	288	3	11	47	46	117	218
	no	1256	529	728	13	21	51	81	186	448
m35	yes	207	68	139	3	7	17	19	64	176
	no	963	572	393	12	16	30	34	116	305
hard	yes	117908	45575	72333	7	8	10	175	211	278
	no	539086	467083	72004	31	37	45	175	208	278

Table 1: Time spent by 2IDA in each part of the method, and number of calls to the sub-problem routine. The second column indicates whether the upper bound from Section 3.5 is used to guide the search, *i.e.*, for splitting and for early stopping. Columns 3-5 provide the total CPU time used for lower bounding and upper bounding calculations. The last 6 column present the minimum, average and maximum number of pricing calls in each part of 2IDA. Recall that UB pricing calls corresponds to both Section 3.5 and the classical CG steps that are run at the end of the method.

run immediately because $\lceil \mathrm{lb}_k \rceil + 1 = \lceil \mathrm{ub}_k \rceil$. The running time dedicated to upper bounding is almost ten times larger than the lower bounding time. The construction of $\mathcal{P}_1$ is not carried out by column generation, since the model has only 2 variables and it is faster to generate directly all constraints. This explains why the number of calls to multichoice knapsack routines is somtimes 0 (columns "LB (2IDA) calls"). In such cases, most of the total 2IDA computing effort is actually spent on proving that this lower bound is optimal.

The 2IDA version with no intermediate upper bounds may be efficient when small instances are considered. The smallest instances `vb10` with $n = 10$ could be solved by using no upper bounding at all. In this case, 2IDA reaches rapidly a state in which all groups have at maximum two elements and the 2IDA lower bound is optimal. This also happens for the smaller `vbInd` instances. Clearly, the results on the most difficult instances are more useful. Therefore, in the remaining we only focus on the version with intermediate upper bounds and we ignore the small instances `vb10`.

6.3 Quality of the Intermediate Dual Bounds

We compare the dual bounds iteratively produced by 2IDA with the Lagrangian dual bounds produced during CG each time the sub-problem is solved. We used the Farley bound, which is a

	Percentage of CG time T_{CG} needed for full convergence									
	5%		10%		20%		30%		40%	
Instance set	CG	2IDA	CG	2IDA	CG	2IDA	CG	2IDA	CG	2IDA
`vb20`	0.41	0.64	0.55	0.95	0.70	0.99	0.85	0.99	0.94	0.99
`vb50-1`	0.29	0.88	0.42	0.97	0.65	0.98	0.79	0.98	0.84	0.99
`vb50-2`	0.49	1.00	0.70	1.00	0.87	1.00	0.97	1.00	0.98	1.00
`vb50-3`	0.44	1.00	0.57	1.00	0.65	1.00	0.88	1.00	0.98	1.00
`vb50-4`	0.52	1.00	0.66	1.00	0.76	1.00	0.94	1.00	0.98	1.00
`vb50-5`	0.52	1.00	0.60	1.00	0.65	1.00	0.79	1.00	0.97	1.00
`vbInd`	0.52	0.64	0.70	0.87	0.71	0.93	0.82	0.93	0.88	0.99
`m01`	0.41	0.94	0.51	0.95	0.65	0.97	0.73	0.97	0.77	0.98
`m20`	0.49	0.74	0.55	0.92	0.66	0.95	0.75	0.98	0.79	0.98
`m35`	0.64	0.47	0.64	0.80	0.69	0.92	0.73	0.96	0.75	0.98
`hard`	0.89	1.00	0.89	1.00	0.89	1.00	0.89	1.00	0.94	1.00

Table 2: Comparison of intermediate dual bounds for CG an 2IDA. We report the average ratio $L_p/\text{OPT}_{\text{CG}}$ for each set of instances for p =5% to p =50% of the time T_{CG}. For example, row `vb20` of column "5%" shows that if both methods are run on instances of data set `vb20`, and stopped after $5\% \cdot T_{\text{CG}}$, CG will output a dual bound equal to 0.41 OPT_{CG}, while 2IDA will output a dual bound equal to 0.64 OPT_{CG}.

well-known specialization of the Lagrangian bound dedicated to this specific model—see Appendix B for the exact formula and references.

Table 2 reports a comparison constructed by the following protocol. We first run the classical CG on each instance. This produces for each instance a reference computing time T_{CG} and an optimal value OPT_{CG}. Then, we run for the same instance both CG and 2IDA with a time limit of $p \cdot T_{\text{CG}}$, where $p \in \{5\%, 10\%, 20\%, 30\%, 40\%\}$. For each such p, we note L_p the best lower bound obtained after $p \cdot T_{\text{CG}}$ time. If 2IDA is stopped during step k, L_p is either the optimum of $\text{DSCP}(\mathcal{P}_{k-1})$ or the best Lagrangian dual bound obtained during step k. Table 2 reports the evolution of $L_p/\text{OPT}_{\text{CG}}$ when p goes from 5% to 40%.

Within $5\%T_{\text{CG}}$ 2IDA finds the CG optimum for all instances `bin3` and `vb50-`$\{2,3,4,5\}$. For the same computing time, the Lagrangian CG lower bounds are equal to respectively 0.89 OPT_{CG}, 0.49 OPT_{CG}, 0.44 OPT_{CG}, 0.52 OPT_{CG}, and 0.52 OPT_{CG} on average.

Within $10\%T_{\text{CG}}$ time, the lower bounds produced by 2IDA are better than 0.92 OPT_{CG} for all instance sets (except `m35`, see below). Withing the same time limit $10\%T_{\text{CG}}$, the Lagrangian GC bounds are at most 0.7 OPT_{CG} except for instance set `bin3`

2IDA slowly closes the gap that is already small after $20\%T_{\text{CG}}$; this is obvious when looking at columns 20% to 40%. When the number of groups becomes larger than 10, 2IDA switches to classical column generation. The same convergence issues occur and even if the Lagrangian bound is almost equal to OPT_{CG}, the method may need time to converge.

The only case in which the classical Lagrangian bound is better is `m35`, only for $p = 5\%$. These instances have all weights larger than $\frac{C}{3}$, and so, all columns have two non-zero coefficients. The first (almost linear) approximations of the dual values are likely to be weak. Note however that after 10% of the time needed for CG, 2IDA becomes more effective in computing dual bounds (0.80 OPT_{CG} on average vs. 0.64 OPT_{CG} for CG).

6.4 Computational Effort Required to Reach the Optimum

The main purpose of 2IDA is to produce high-quality dual bounds in a faster way than the classical CG. However, the fact that almost optimal dual bounds are produced in the first iterations hints that the method can help proving optimality in a faster way.

Table 3 compares the CPU time measures and the number of sub-problems calls needed to

	Avg. CPU [ms]		2IDA vs CG (CPU)			nb sub-problems	
Instance set	2IDA	CG	$\prec$	$\simeq$	$\succ$	2IDA	CG
vb20	9161	11653	10	4	11	89	63
vbIndstr	7580	14252	10	1	6	43	51
vb50-1	39827	50164	14	2	4	179	125
vb50-2	89432	125399	17	0	3	206	223
vb50-3	44068	106356	20	0	0	80	159
vb50-4	66498	87604	19	1	0	242	199
vb50-5	28490	85680	20	0	0	62	160
m01	818	849	549	43	408	161	191
m20	423	620	793	49	158	128	173
m35	207	379	920	42	38	71	134
hard	117908	376112	10	0	0	219	768

Table 3: The computing effort for final convergence with 2IDA and CG. Columns "Avg. CPU" report the time (in ms.). Columns "2IDA vs CG (CPU)" report the number of times 2IDA is (1) faster than CG (column $\prec$), (2) roughly equally fast (difference $< 5\%$, column $\simeq$), and (3) slower (column $\succ$). The last two columns compares the total number of pricing calls for 2IDA and CG. For 2IDA, the total number of pricing calls considers both Multiple-Choice Knapsack problems (columns of $\mathcal{P}_k$) and classical Knapsack problems (columns used for upper bounding, including the inherited columns from Section 3.6).

converge by either method. 2IDA is significantly faster for 2402 instances out of 3127 (76%), and at least as good in 2544 cases (81%). On average, our method is faster than CG for all data sets. This can be explained by the fact that less computationally demanding (aggregated) pricing procedures are run using 2IDA. The first iterations are more effective for producing a good dual bound (as shown above). Furthermore, by only generating $\mathbf{y}^*_\mathbf{k}$-*tight constraints* during the 2IDA upper bounding process (Section 3.5), we introduce a form of implicit stabilization around those good initial solutions.

Note that the most difficult instances are `hard`. For these instances, our approach is always faster (a factor 3 on average) and produces less sub-problems.

To conclude, we would like to give two essential keys for the efficiency of the method. First, dual-feasible functions are useful to make good split decisions. This means that knowing *a priori* the structure of a good dual solution can be useful in 2IDA. Secondly, the incremental construction of polytope $\mathcal{P}_{k+1}$ from $\mathcal{P}_k$ (Section 3.6) reduces the computing time by a wide range compared to constructing $\mathcal{P}_{k+1}$ from scratch at each iteration. This part should not be overlooked in a good implementation of 2IDA: in certain cases, this inheritance process allows 2IDA to compute lb_{k+1} from lb_k in virtually no time (see detailed instance-by-instance results at `www.lgi2a.univ-artois.fr/~porumbel/cs/details2ida.zip`).

6.5 Cutting Stock Problem with Maximum Waste

Since 2IDA is not dedicated to the cuttig-stock problem and its structure, we now consider the Cutting Stock Problem with Maximum Waste (CSP-MW). The problem is defined as follows: all valid patterns are required to have a total length (weight) between C^- and C^+. In practice, one can consider that $C^+ = C$ is the roll length and $C^+ - C^-$ is the maximum allowed waste. This constraint arises in industry when large pieces of waste cannot be recycled. The new maximum waste constraints might make certain items i be cut more than b_i times (such additional cuts are not considered as waste).

It is interesting to consider this problem, because there is no equivalent of the dual-feasible functions for this variant, and thus we have no information about the structure of good dual solutions.

We consider the same instances from the previous sections. For each benchmark set, we report individual results for the first three instances. The maximum waste is set as follows:

Instance	k=1		k=2		k=3		CG_{OPT}
	lb_{2IDA} [T_{ms}]	lagr [T_{ms}]	lb [T_{ms}]	lagr [T_{ms}]	lb [T_{ms}]	lagr [T_{ms}]	CS-MW/CS
m01-1	49 [20]	52 [30]	53 [30]	49 [40]	53 [60]	35 [70]	54/54
m01-2	infeasible for current conditions ($C^- = C^+ = C = 100$)						
m01-3	50 [10]	24 [20]	54 [30]	47 [40]	54 [50]	41 [60]	55/54
vb50-1p01	940*[960]	418 [1090]	940*[1870]	572 [2010]	950*[2490]	571 [2560]	1219/939
vb50-1p02	896 [1670]	329 [1750]	898 [3080]	508 [3320]	905*[4260]	537 [4480]	1000/898
vb50-1p03	989*[1200]	319 [1250]	989*[2240]	528 [2360]	989*[3070]	473 [3120]	1200/928
vb50-2p01	731 [1740]	266 [1890]	731 [2820]	405 [3010]	731 [4370]	585 [4550]	754/737
vb50-2p02	679 [2140]	337 [2260]	679 [3470]	463 [3510]	679 [5190]	453 [5280]	682/679
vb50-2p03	560 [2040]	288 [2130]	560 [3490]	327 [3680]	560 [6100]	358 [6160]	560/560
vb50-3p01	329 [2660]	103 [2730]	329 [5280]	157 [5490]	329 [5690]	170 [5840]	329/329
vb50-3p02	280 [2810]	104 [2840]	280 [5790]	148 [5820]	280 [10890]	139 [11060]	280/280
vb50-3p03	317 [2740]	138 [3050]	317 [6160]	181 [6260]	317 [7070]	173 [7190]	317/317
vb50-4p01	673 [1440]	282 [1500]	673 [2440]	371 [2470]	673 [2560]	392 [2680]	683/673
vb50-4p02	640 [1520]	331 [1540]	640 [2500]	378 [2530]	640 [2710]	340 [2720]	659/640
vb50-4p03	774 [1060]	380 [1180]	774 [1720]	444 [1870]	774 [2040]	559 [2210]	861/775
vb50-5p01	394 [1900]	216 [2060]	394 [3990]	240 [4000]	394 [4850]	173 [5080]	394/394
vb50-5p02	408 [1860]	244 [1980]	408 [4780]	193 [4940]	408 [5510]	132 [5570]	408/408
vb50-5p03	345 [2100]	179 [2320]	345 [4380]	193 [4500]	345 [4860]	182 [4940]	345/345
vbInd30p0	90 [260]	21 [270]	90 [540]	20 [560]	90 [950]	73 [970]	90/90
vbIndd43p20	29 [20]	15 [50]	29 [40]	15 [50]	29 [60]	13 [80]	36/29
vbIndd43p21	40 [90]	24 [160]	40 [270]	33 [320]	40 [1200]	9 [1240]	40/40
hard-1	56 [520]	47 [620]	56 [214160]	60[162190]	60 [295850]	60[162190]	60/57
hard-2	56 [530]	47 [630]	56 [157090]	58[144710]	58 [224760]	58[144710]	58/56
hard-3	55 [540]	47 [620]	55 [174680]	59[163040]	58 [234420]	59[163040]	59/55

Table 4: The first column indicates the instance name. Columns 2, 4 and 6 respectively provide the bound obtained by 2IDA for k from 1 to 3. Columns 3, 4, and 5 respectively report the Lagrangian bound obtained using a similar CPU time as 2IDA. More exactly, for each k, we report the best Lagrangian bound found during CG at the first moment when the CG running time is larger than the one needed by 2IDA. The last two columns indicate the CG optimum with and respectively without maximum waste.

* Remark that these 2IDA bounds are better than the CG optimum in the last column. Any bound for the standard Cutting Stock Problem (*e.g.*, based on DFF) could never reach this value, as they would all be limited by the Cutting-Stock optimum.

- 0 for the `m01` instances (*i.e.*, $C^- = C^+ = C = 100$). The instances `m20` and `35` were not used because they contain only large items (at least $\frac{20}{100}C$ or respectively $\frac{35}{100}C$, see Appendix A), and so, they are most often infeasible;
- 2 for all `vb50` instances (*i.e.*, $C^- = 99998$);
- $0.5\%C$ for `vbInd` instances; for this set, we provide results for the only three instances with $n \geq 30$;
- 4, for the `hard` instances (*i.e.*, $C^- = 196$).

Table 4 compares the bounds obtained by 2IDA without upper bounding and with splitting decisions taken using weight spread indicators (as described in Section 3.4.1). The conclusions are clear: the 2IDA bounds clearly dominate the Lagrangian bounds. More exactly, even the first 2IDA bounds for $k = 1$ (Column 2) are usually larger than the Lagrangian bounds reported after 2-3 times more time (Column 7). The only exception arises for the hard instances, where CG converges more rapidly.

7 Conclusions and Perspectives

We described an aggregation method for computing dual bounds in column generation in a way that is faster than classical Lagrangian bounds. The new method proceeds by constructing an iterative inner approximation of the dual polytope and converges towards the optimum of the original model. Several computational techniques are used to incrementally construct these inner polytopes and obtain a practically effective method. Computational results show that, besides generating fast dual bounds, the method can reach the optimum of the original model more rapidly than classical column generation in many cases.

Future research work will concentrate on the case with several resources, and also more general cases where the pricing sub-problems are solved by an ILP solver.

References

[1] H. B. Amor, J. Desrosiers, and J. M. V. de Carvalho. Dual-optimal inequalities for stabilized column generation. Operations Research, 54(3):454–463, 2006.

[2] H. Ben Amor and J. M. V. de Carvalho. Cutting stock problems. In G. Desaulniers, J. Desrosiers, and M. Solomon, editors, Column Generation, pages 131–161. Springer US, 2005.

[3] P. Benchimol, G. Desaulniers, and J. Desrosiers. Stabilized dynamic constraint aggregation for solving set partitioning problems. European Journal of Operational Research, 2012.

[4] O. Briant, C. Lemarchal, P. Meurdesoif, S. Michel, N. Perrot, and F. Vanderbeck. Comparison of bundle and classical column generation. Mathematical Programming, 113(2):299–344, 2008.

[5] A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. Discrete Applied Mathematics, 111:231–262, 2001.

[6] J. Carvalho. Using extra dual cuts to accelerate column generation. INFORMS Journal on Computing, 17(2):175–182, 2005.

[7] F. Clautiaux, C. Alves, and J. Carvalho. A survey of dual-feasible and superadditive functions. Annals of Operations Research, 179(1):317–342, 2009.

[8] F. Clautiaux, C. Alves, J. M. V. de Carvalho, and J. Rietz. New stabilization procedures for the cutting stock problem. INFORMS Journal on Computing, 23(4):530–545, 2011.

[9] O. Du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized column generation. Discrete Mathematics, 194:229–237, 1999.

[10] I. Elhallaoui, A. Metrane, F. Soumis, and G. Desaulniers. Multi-phase dynamic constraint aggregation for set partitioning type problems. Mathematical Programming, 123(2):345–370, 2010.

[11] I. Elhallaoui, D. Villeneuve, F. Soumis, and G. Desaulniers. Dynamic aggregation of set-partitioning constraints in column generation. Operations Research, 53(4):632–645, 2005.

[12] M. Gendreau, G. Laporte, and F. Semet. Heuristics and lower bounds for the bin packing problem with conflicts. Computers and Operations Research, 31:347–358, 2004.

[13] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem. Operations Research, 9:849–859, 1961.

[14] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem - part II. Operations Research, 11:863–888, 1963.

[15] L. Kantorovich. Mathematical methods of organizing and planning production. Management Science, 6:363–422, 1960.

[16] I. Litvinchev and V. Tsurkov. Aggregation in large-scale optimization, volume 83 of Applied Optimization. Springer, 2003.

[17] M. Luebbecke and J. Desrosiers. Selected topics in column generation. Operations Research, 53:1007–1023, 2005.

[18] G. S. Lueker. Bin packing with items uniformly distributed over intervals $[a, b]$. In 24th Annual Symposium on Foundations of Computer Science, pages 289–297. IEEE, 1983.

[19] R. Macedo, C. Alves, J. M. V. de Carvalho, F. Clautiaux, and S. Hanafi. Solving the vehicle routing problem with time windows and multiple routes exactly using a pseudo-polynomial model. European Journal of Operational Research, 214(3):536–545, 2011.

[20] R. Marsten, W. Hogan, and J. Blankenship. The BOXSTEP method for large-scale optimization. Operations Research, 23(3):389–405, 1975.

[21] M. G. N. Azi and J.-Y. Potvin. An exact algorithm for a single-vehicle routing problem with time windows and multiple routes. European Journal of Operational Research, 178:755–766, 2007.

[22] A. Oukil, H. B. Amor, J. Desrosiers, and H. E. Gueddari. Stabilized column generation for highly degenerate multiple-depot vehicle scheduling problems. Computers and Operations Research, 34(3):817–834, 2007.

[23] R. Sadykov and F. Vanderbeck. Column generation for extended formulations. Electronic Notes in Discrete Mathematics, 37:357–362, 2011.

[24] A. Scholl, R. Klein, and C. Jurgens. BISON: a fast hybrid procedure for exactly solving the one-dimensional bin-packing problem. Computers and Operations Research, 24:627–645, 1997.

[25] C. Shetty and R. Taylor. Solving large-scale linear programs by aggregation. Computers and Operations Research, 14(5):385–393, 1987.

[26] F. Vanderbeck. Computational study of a column generation algorithm for bin packing and cutting stock problems. Mathematical Programming, 86(3):565–594, 1999.

[27] F. Vanderbeck and M. W. P. Savelsbergh. A generic view of dantzig-wolfe decomposition in mixed integer programming. Operations Research Letters, 34(3):296–306, 2006.

[28] M. V. Vyve and L. A. Wolsey. Approximate extended formulations. Mathematical Programming, 105(2-3):501–522, 2006.

A Instance Sets and Computing Setting

All reported CPU times are obtained using the following computing environment: a HP ProBook 4720 laptop clocked at 2.27GHz (Intel Core i3), with the `gnu g++` C++ compiler on Linux Ubuntu, kernel version 2.6. The master problems are solved using Cplex 12.3 and Ilog Concert libraries. The source code and the instances are publicly available on-line at `www.lgi2a.univ-artois.fr/~porumbel/cs/`. Detailed results (instance by instance) are also publicly available at this address.

We now describe in details the instances we used in this paper. They form a set of 3127 individual instances. The number of items ranges from 10 to 200 and the capacities is between 100 and 100000.

`VB-a` instances `vb10`, `vb20` (random instances [26]) and `vbIndst` (industrial instances [26]). We selected only the industrial instance with one bin type;

`VB-b` instances `vb50-1`, `vb50-2`,..., `vb50-5`, random instances;

`BP` three randomly generated bin-packing instance sets [7]: `m01`, `m20`, `m35`;

`hard` 10 bin-packing instances [24] long-acknowledged for their difficulty in the bin-packing literature

Name	n	C	avg. $\mathbf{b}$ span	avg. $\mathbf{w}$ span	Description
`vb10`	10	10000	$[10, 100]$	$[1, \frac{1}{2}C]$	20 random instances [26]: `CSTR10b50c[1-5]*` files[a]
`vb20`	20	10000	$[10, 100]$	$[1, \frac{1}{2}C]$	25 random instances [26]: `CSTR20b50c[1-5]*` files[a]
`vb50-c1`	50	10000	$[50, 100]$	$[1, \frac{3}{4}C]$	20 random instances [26]: `CSTR50b50c1*` files[a]
`vb50-c2`	50	10000	$[50, 100]$	$[1, \frac{1}{2}C]$	20 random instances [26]: `CSTR50b50c2*` files[a]
`vb50-c3`	50	10000	$[50, 100]$	$[1, \frac{1}{4}C]$	20 random instances [26]: `CSTR50b50c3*` files[a]
`vb50-c4`	50	10000	$[50, 100]$	$[\frac{1}{10}C, \frac{1}{2}C]$	20 random instances [26]: `CSTR50b50c4*` files[a]
`vb50-c5`	50	10000	$[50, 100]$	$[\frac{1}{10}C, \frac{1}{4}C]$	20 random instances [26]: `CSTR50b50c5*` files[a]
`vb50-b100`	50	10000	$[1, 210]$	$[\frac{1}{10}C, \frac{1}{2}C]$	20 random instances [26]: `CSTR50b100c4*` files[a]
`m01`	100	100	1	$[1, C]$	1000 random `bin-packing` instances [7];
`m20`	100	100	1	$[\frac{20}{100}C, C]$	1000 random `bin-packing` instances [7];
`m35`	100	100	1	$[\frac{35}{100}C, C]$	1000 random `bin-packing` instances [7];
`Hard`	≈ 200	100000	$1 - 3$	$[\frac{20}{100}C, \frac{35}{100}C]$	Bin-packing instances, known to be more difficult

Table 5: General characteristics of the instances. Columns 4 and 5 indicate the (approximate) interval of the demand value, and, respectively, item weights.

[a] In the archive `http://www.math.u-bordeaux1.fr/~fvanderb/data/randomCSPinstances.tar.Z`

B The Lagrangian Bound for CSP and CSP-MW

We now recall the specialized Lagrangian lower bound used for the standard Cutting-Stock Problem and that can be used for the Cutting-Stock Problem with Maximum Waste. We come back to the main Set-Covering column generation model (2.1)-(2.2), p. 3. Similarly to what is done in [2, § 2.2], [26, §. 3.2], [17, §. 2.1] or [4, § 1.2], Lagrangian bounds are constructed as follows. Consider a given iteration of the CG process with dual values $\mathbf{y}$. We use these values $\mathbf{y}$ as multipliers of the Lagrangian relaxation of (2.1). The relation between the CG optimum $\mathrm{OPT}_{\mathrm{CG}}$ and the Lagrangian bound $\mathcal{L}_{\mathbf{y}}$ can be written:

$$\mathrm{OPT}_{\mathrm{CG}} \geq \mathcal{L}_{\mathbf{y}} = \min_{\lambda \geq \mathbf{0}} \left(\sum_{\mathbf{a} \in \mathcal{A}} (1 - \mathbf{a}^\top \mathbf{y}) \lambda_a \right) + \mathbf{b}^\top \mathbf{y} \geq \min_{\lambda \geq \mathbf{0}} \left(M_{\mathrm{RC}} \sum_{\mathbf{a} \in \mathcal{A}} \lambda_a \right) + \mathbf{b}^\top \mathbf{y}, \tag{B.1}$$

where: λ is the primal variable vector of the column generation, and $M_{\mathrm{RC}} = \min_{\mathbf{a} \in \mathcal{A}} 1 - \mathbf{a}^\top \mathbf{y} \leq 0$ is the *non-positive* minimum reduced cost at the current iteration, The key for calculating the

Farley bound resides in observing that the inequality below can be artificially added to the initial LP.

$$\sum_{\mathbf{a}\in\mathcal{A}} \lambda_a \leq \mathrm{OPT}_{\mathrm{CG}}, \tag{B.2}$$

The 2IDA Lagrangian bounds from Section 6.3 are calculated in the same manner, *i.e.*, the whole reasoning still holds by only using $\mathcal{A}_k$ instead of $\mathcal{A}$. Combining (B.2) with $M_{\mathrm{RC}} \leq 0$, we can reduce (B.1) to $\mathrm{OPT}_{\mathrm{CG}} \geq M_{\mathrm{RC}} \cdot \mathrm{OPT}_{\mathrm{CG}} + b^\top \mathbf{y}$. The Farley lower bound $\mathcal{L}^F_{\mathbf{y}}$ follows immediately: $\mathcal{L}^F_{\mathbf{y}} = \frac{b^\top \mathbf{y}}{1-M_{\mathrm{RC}}} \leq \mathrm{OPT}_{\mathrm{CG}}$.